Performance Aspects of Synthesizable Computing Systems

Schleuniger, Pascal

Publication date:
2014

Document Version
Publisher's PDF, also known as Version of record

Performance Aspects of Synthesizable Computing Systems

Pascal Schleuniger

Kongens Lyngby 2014
COMPUTE-PHD-2014-337

Technical University of Denmark
DTU Compute
Matematiktorvet, Bygning 303 B, DK-2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

COMPUTE-PHD: ISSN 0909-3192

Summary

Embedded systems are used in a broad range of applications that demand high performance within severely constrained mechanical, power, and cost requirements. Embedded systems implemented in ASIC technology tend to provide the highest performance, lowest power consumption and lowest unit cost. However, high setup and design costs make ASICs economically viable only for high volume production. Therefore, FPGAs are increasingly being used in low and medium volume markets. The evolution of FPGAs has reached a point where multiple processor cores, dedicated accelerators, and a large number of interfaces can be integrated on a single device.

This thesis consists of five parts that address performance aspects of synthesizable computing systems on FPGAs. First, it is evaluated how synthesizable processor cores can exploit current state-of-the-art FPGA architectures. This evaluation results in a processor architecture optimized for a high throughput on modern FPGA architectures. The current hardware implementation, the Tinuso I core, can be clocked as high as 376MHz on a Xilinx Virtex 6 device and consumes fewer hardware resources than similar commercial processor configurations. The Tinuso architecture leverages predicated execution to circumvent costly pipeline stalls due to branches and exposes hazards to the compiler to keep the hardware simple. Second, it is investigated if a production compiler, GCC, is able to successfully leverage predicated execution and schedule instructions so as to mitigate the hazards. The third part of this thesis describes the design and implementation of communication structures for Tinuso multicore configurations and evaluates the scalability of these systems. Forth, a case study shows how to map a high performance synthetic aperture radar application to a synthesizable multicore system. The proposed system includes 64 processor cores and a 2D mesh interconnect on a single FPGA device and consumes about 10 watt only. Finally, a task based programming model is proposed that allows for easily expressing parallelism and simplifies memory management.

Resumé

Indlejrede systemer anvendes i dag i en lang række applikationer der kræver høj ydeevne, men som er underkastet skarpe restriktioner i henhold til mekanisk design, strømforbrug og pris. Indlejrede systemer der er implementeret ved hjælp ASIC-teknologi opnår typisk den højeste ydeevne med det laveste strømforbrug og den laveste enhedspris. En høj indgangspris kombineret med en lav enhedspris gør ASIC-teknologi egnet til masseproduktion, men uegnet til små produktionsmængder. Derfor bliver FPGA teknologi i stigende grad anvendt i markeder med små produktionsmængder. FPGA-teknologien er efterhånden blevet forfinet til et sådant niveau, at mange-kernede processor systemer, dedikerede acceleratorer og et stort antal interfaces kan implementeres på en enkelt FPGA enhed.

Denne afhandling består af fem dele. Afhandlingen behandler og undersøger ydeevnen for syntetiserebare computersystemer på FPGA-enheder. I første del af afhandlingen evalueres måder hvorpå syntetiserebare processorkerner kan udnytte de nyeste avancerede FPGA-arkitekturer. Denne evaluering resulterer i en processorarkitektur der er optimeret til at opnå høj ydeevne på moderne FPGA-enheder. Den nuværende implementering af denne processorarkitektur, kaldet Tinuso-I, kan køres med en taktfrekvens på op til 376MHz på en Xilinx Virtex 6 enhed. Tinuso-I anvender færre hardware ressourcer end andre kommercielle processorer i sin klasse. Tinuso arkitekturen anvender prædikeret eksekvering for at undgå dyre pipeline stall forsaget af hop i instruktionsstrømmen. Tinuso arkitekturen eksponerer pipeline hazards til compileren for at holde hardwaren simpel.

I anden del af afhandlingen unsersøges det hvorvidt en produktionskvalificeret compiler, GCC, er i stand til at anvende prædikerede instruktioner og tilrettelæggelse af instruktionsstrømmen for at mindske effekten af pipeline hazards.

Tredje del af afhandlingen beskriver design og implementering af kommu-

nikationsstrukturerne for flere Tinuso multikerne konfiguratioer og evaluerer skalerbarheden af de resulterende systemer.

Fjerde del af afhandlingen er et casestudie der viser hvordan en højtydende syntetisk apparatur radar applikation kan afvikles på et syntetiserebart multikerne system. Det anvendte system består af 64 processorkerner og et 2D kommunikationsnetværk der implementeres på en enkelt FPGA enhed. Systemet bruger omkring 10 watt.

I sidste del af afhandlingen præsenteres en jobbaseret programmeringsmodel der simplificerer hukommelsesmanagement og gør det nemt at udtrykke parallelisme.

Preface

This thesis was prepared at DTU Compute, at the Technical University of Denmark in partial fulfillment of the requirements for acquiring the Ph.D. degree in engineering.

The Ph.D.-project was supervised by Associate Professor Sven Karlsson and Professor Jan Madsen.

Lyngby, April 2014

Pascal Schleuniger

Papers included in the thesis

I Pascal Schleuniger, Sven Karlsson. A Synthesizable Multicore Platform for Microwave Imaging. *Proceedings of the 10th International Symposium on Applied Reconfigurable Computing ARC*, 2014. Presented and published.

II Pascal Schleuniger, Anders Kusk, Jørgen Dall, Sven Karlsson. Synthetic Aperture Radar Data Processing on an FPGA Multi-Core System. *Proceedings of the 25th International Conference on Architecture of Computing Systems ARCS*, 2013. Presented and published.

III Pascal Schleuniger, Sally A. McKee, Sven Karlsson. Design Principles for Synthesizable Processor Cores. *Proceedings of the 25th International Conference on Architecture of Computing Systems ARCS*, 2012. Presented and published.

IV Pascal Schleuniger, Nicklas Bo Jensen, Sven Karlsson. A Compiler Infrastructure for a High Performance Synthesizable Processor Core. Workshop article, manuscript ready for submission.

V Pascal Schleuniger, Anders Kusk, Jørgen Dall, Sven Karlsson. Synthetic Aperture Radar Data Processing on an FPGA Multi-Core system. Journal article, manuscript ready for submission.

Acknowledgments

There is a long list of persons I want to thank for their support during the work on this thesis. First of all I thank my advisor Sven Karlsson for his guidance, support, motivation, critic, and many interesting discussions. I thank Professor Jan Madsen and Michael Reibel Boesen who have introduced me into the world of research. I thank Professor Thomas Gross who enabled me a research stay in his group at ETH in Zurich. I thank Karin Tunder for her help in the administrative jungle. I also thank Nicklas Bo Jensen, Laust Brock-Nannestad, and Andreas Hindborg for their work on GCC and Binutils. Finally, I thank my wife Gabi and my two boys, Alexander and Kim, for their mental support and for their patience whenever I was travelling or mentally absent while writing this thesis.

Contents

Chapter 1

Introduction

Embedded systems are computing systems that are designed for specific applications. A broad range of applications demand high performance within severely constrained mechanical, power, and cost requirements. Embedded systems may either be composed of discrete processor cores and interface devices or implemented as dedicated hardware. Systems implemented in application specific integrated circuits, *ASICs*, tend to provide the highest performance, lowest power consumption and lowest unit cost. However, design effort and setup production costs for ASICs are high and increasing as process technology evolves. High development costs make ASICs economically viable for high volume production only. Field programmable gate arrays, *FPGAs*, on the other hand, have a higher unit cost but essentially no setup cost. Due to this, FPGAs are increasingly being used in low and medium volume markets.

The inherently parallel structure of FPGAs allows for an efficient implementation of parallel algorithms. Sequential algorithms, on the other hand, are easier to implement on a microprocessor. However, for many applications it is convenient to have both a microprocessor to execute sequential tasks and an FPGA to accelerate parallel sections of an algorithm. Of course, one can compose embedded systems of discrete microprocessors and FPGA devices but they also can be combined in one chip using synthesizable processor cores. Computer systems with integrated synthesizable processor cores tend to have a simpler board layout, fewer problems with signal integrity and electromagnetic interference, and reduced system costs.

The performance of synthesizable processor cores tends to be significantly lower than discrete microprocessors. Current synthesizable processor cores are

primarily designed for configurability rather than for raw performance. Major FPGA vendors such as Xilinx and Altera offer in-order, single issue processor cores optimized for their technology but do not exploit current state-of-the-art FPGA architectures. For example, neither Xilinx MicroBlaze nor Altera Nios II make use of pipelined memory resources.

One possibility to improve the performance of synthesizable processor cores is to optimize the hardware for a high clock frequency. Pipelined memory resources found in modern state-of-the-art FPGA architectures allow for a fast implementation of caches and register file. However, this requires breaking pipeline stages into smaller stages. While this results in a design with a significantly higher clock frequency it breaks the compatibility to existing instruction set architectures. Moreover, branch instructions become expensive, as branch address and direction are resolved late in the pipeline. For example, in processor designs with pipelined caches and register file at least four successive instructions are fetched until the branch address is computed. These instructions are called *delay slots* that the compiler has to attempt to fill with useful independent instructions. If unsuccessful, the compiler will have to insert no-operation, *nop*, instructions which will reduce performance.

One possibility to reduce branch costs is to leverage predicated instructions. A predicated instruction is executed if a condition that is specified in the operation code is true, otherwise the instruction is annulled and has no effect. Apart from advantages in filling delay slows, predicated instructions allow a compiler to replace some conditional control-flow sequences with predicated instructions and hence avoid delay slots. However, leveraging predicated execution increases the complexity in a compiler.

The performance of a single issue in-order processor might not be sufficient for many embedded application. Out-of-order and superscalar executions have proven to be very effective techniques to extract parallelism at instruction level. Dynamic scheduling with register re-naming effectively reduces data hazards. This comes at the cost of complex multiported structures that do not perform well on FPGA architectures. Alternatively it seems more feasible to improve performance of FPGA computing by extracting parallelism at task level. The evolution of FPGAs has reached a point where multiple processor cores, dedicated accelerators, and a large number of interfaces can be integrated on a single device. However, efficient communication structures are required to successfully extract parallelism in multicore and many core systems. For in-order processor cores it is difficult to hide memory latency. Therefore an intercommunication network has to be optimized for a lowest possible latency and high clock frequency to attain a high throughput.

Considering the logic integration of the newest generation FPGAs it is possible to integrate hundreds of processor cores into a single device. Such reconfigurable multicore systems open up new possibilities for high performance applications. For example, active microwave imaging techniques such as radar and tomography are used in a wide range of medical, industrial, scientific, and military applications. Microwave imaging systems consists of synchronized radio transmitters and receivers that emits radio waves and process their reflections to reconstruct an image of an object. Often a very high number of operations is required to reconstruct the output image because each pixel must analyze hundreds of these reflections.

Given the large amount of data and the parallel structure of image reconstruction applications, graphic processing units, GPUs, are well suited for this type of data processing. However, many applications come with strict power requirements, limitations on system size, require the use of industrial and space grade components, or demand a certain durability and reliability of the system. For these applications, off-the-shelve GPUs may not be an appropriate solution. Instead, a multicore system on FPGA is a convenient alternative. As many modern microwave imaging system successfully use FPGAs for signal processing already it is evident to integrate data processing functionality into the existing device.

The use of a multicore system raises the abstraction level for the application programmer without facing the current performance drawbacks of high-level synthesis. Moreover, mapping an application to multicore system significantly reduces development effort over a fully custom FPGA implementation.

Finally, there must be found ways to program the system. It may be conceptually trivial to imagine multicore systems but it remains a challenge to program these systems. Researchers aim for programming languages, programming models, and runtime systems that support the programmer expressing parallelism and exploiting parallel hardware architectures.

The problem statement of this thesis is to evaluate architectural trade-offs to improve the performance of synthesizable computing systems on FPGAs. In particular, this thesis addresses the following research questions:

- Instruction level parallelism: The performance of a computer system can be improved by exploiting parallelism. In a single processor system instruction level parallelism is applied to perform multiple operations simultaneously. The internal structure of FPGAs is well suited for pipelining, i.e., splitting up the critical path of a design into several combinatorial-flip-flop sections. Superpipelining breaks pipeline stages into smaller stages

and leverages instruction level parallelism by executing operations partially overlapped. Shorter pipeline stages reduce the clock period and thus allow for a higher clock rate and instruction throughput. However, superpipelining cannot be applied arbitrarily, additional hardware resources and increasing complexity in the control logic lead to a diminishing return. It is not evident if superpipelining can pay-off for synthesizable processor cores on modern FPGA architectures.

- Predicated execution: The performance of a computer system depends on the implementation of control flow instructions. For example, branch instructions in highly pipelined systems are costly as branch address and branch direction are computed late in the pipeline. Implementations that flush or stall the pipeline reduce the level of instruction level parallelism. Predicated execution is a technique to eliminate control dependencies and thus, reduces the number of branch instructions in a program. Predicated instructions have an additional predication operand, called predicate. Depending on the value of the predicate the instruction is executed or annulled, i.e., has no effect. Predicated instructions can therefore be used to replace simple if-statements. Moreover, predicated instructions can be used to fill delay slots. While the concept of predicated execution is simple, it is a challenge for a compiler to exploit predicated execution. Moreover, predicated execution takes up space in the instruction word encoding and increases the hardware complexity. Hence, it is not clear whether the efforts to implement predicated execution on a synthesizable processor core will be worth it.

- Scalability: Multicore systems often exploit parallelism at task level. Each processor executes a different sequence of a program, called task. Tasks access memory and communicate with each other through an intercommunication network. The logic integration of modern FPGAs has reached a point where hundreds of processor cores can be integrated on a single device. However, intercommunication network and memory are shared resources of a system and may therefore limit the scalability. Moreover, the scarce on-chip memory resources of an FPGA may be a limiting factor for the memory-system performance in large-scale systems. Hence, it is highly application dependent to which extent synthesizable computing systems scale.

- Programmability: A parallel programming model is an abstraction of a computer system including hardware, compiler, and system software that

enables the expression, compilation, and execution of parallel applications. Programming models aim to provide the programmer with the highest level of abstraction to express parallelism that yet enables an efficient mapping of applications to the hardware architecture. Commonly used functionality of a programming model may be implemented in hardware to improve performance. For example, hardware primitives may enable efficient communication or simplify memory resource management. With the advent of multicore systems, the main bottleneck shifts from computation capabilities to data management. A key point of all multicore architectures is therefore the memory hierarchy. Minor restrictions in the programming model may simplify memory consistency and enable more efficient communication and cache architectures. Work on scalable programming models is an active area of research that involves many trade-offs. As efficient solutions highly depend on the platform, a programming model optimized for the proposed synthesizable multicore system is required.

The aim of this thesis is to research these architectural trade-offs and to apply them in the design of a synthesizable multicore system.

1.1 Contributions

The work leading to this thesis includes the following contributions:

- The main contribution is the design of the Tinuso processor architecture. Tinuso exploits hardware resources found on modern state-of-the-art FPGAs. Superpipelining allows for a significantly higher system clock frequency. The instruction set architecture is designed to enable predicated execution of all instructions. Predicated instructions are used to reduce the high branch cost of the deep pipeline. To keep the design simple all types of hazards are exposed to software. The current prototype can be clocked as high as 376MHz on a Xilinx Virtex 6 and consumes fewer hardware resources than similar commercial processor configurations. The performance is evaluated by running a set of numerical and search-based assembly-benchmarks where Tinuso achieves an average performance improvement of 38% over a similar Xilinx MicroBlaze configuration.

- A GCC compiler backend for the Tinuso architecture was evaluated. This work includes a number of architecture specific scheduling optimizations

in the compilation toolchain that allow for resolving hazards in software. The compiler backend is evaluated by analyzing the code execution time, instruction mix and the number of executed predicated instructions of a set of small C benchmarks. Tinuso shows an average performance improvement of 56% over a similar Xilinx MicroBlaze configuration for set of small C benchmarks.

- A generic packet switched, XY routed, 2D mesh on-chip network optimized for FPGA implementation was designed. This work includes optimizations in routing scheme and flow control mechanism to attain high clock frequency and low latency. The feedback loop to manage contention is pipelined to reduce the routing delay in the time-critical path of the design. The network can be clocked with up to 300 MHz with a peak switching data rate of 9.6 Gbits/s per link on state-of-the-art FPGAs.

- In a case study it was demonstrated how to successfully map data processing for the POLARIS synthetic aperture radar to a multicore system with 64 Tinuso processor cores. The direct back-projection application provides real-time data processing for a 3000m wide area with a resolution of 2x2 meters. The multicore fabric consisting of 64 processor cores and 2D mesh network-on-chip utilizes 60% of the hardware resources of a Xilinx Virtex-7 device with 550 thousand logic cells and consumes about 10 watt.

- A proposal of a programming model was presented that allows for easily expressing task level parallelism. The programmer defines parallel tasks with fork–join primitives that are bound to a group of processor cores, which execute these tasks concurrently. A work stealing policy is proposed to achieve an optimal load balancing. The programming model is targeted to heterogeneous multicore systems, hence, it is possible to bind tasks to specific processor cores. The runtime system includes hardware primitives that implement task queue, which reduces memory traffic and avoids data races when multiple cores access a task. Moreover, minor restrictions in the programming model are proposed that enable a lightweight implementation of a hardware cache coherency mechanism.

- Tinuso hardware structures and toolchain can be used as a cycle accurate system simulator. Tinuso's VHDL sources are fully synthesizable to both Altera and Xilinx FPGAs. Hence, there is a broad range of tools available to simulate Tinuso systems. The work for this thesis includes a

memory controller, simple I/O infrastructure, and a performance monitoring framework. Performance counters monitor output network utilization, cache misses, processor idle time, and the instruction mix of an executed application. The platform can be synthesized and run on hardware to obtain a high simulation speed. Alternatively, Tinuso multicore systems can be simulated with a very high level of detail, which is useful for example for debugging.

1.2 Thesis Outline

The following briefly outlines the structure of the thesis. Background information and descriptions of related research is scattered across all chapters of the thesis.

- Chapter 2 defines some key terms that will be used throughout the rest of this thesis.
- Chapter 3 introduces the architecture of Tinuso and describes design and implementation of the synthesizable processor core. An evaluation of the Tinuso processor core in terms of clock speed, utilized hardware resources and code execution time of a set of integer micro-benchmark is included in this chapter.
- Chapter 4 introduces the GNU toolchain that consists of C compiler and Binutils, a collection of low level programming tools. This chapter describes the implementation of a GCC compiler toolchain for the Tinuso architecture including architecture specific scheduling optimizations and reports how to leverage predicated execution. The compiler backend for Tinuso is evaluated with a set of small C benchmarks.
- Chapter 5 describes design and implementation of a generic 2D mesh on-chip interconnect. The network is optimized for a latency of one cycle per network hop and a high clock frequency. The network is evaluated by measuring the network latency of random traffic.
- Chapter 6 considers Tinuso multicore systems to be used as simulation platforms for research on multicore systems and programming models. The design and implementation of communication interface that allows for explicit communication with other processing elements is described as well as the interfaces of Tinuso's building blocks. Research platforms may

require a very high number of processor cores. Therefore the hardware scalability of Tinuso multicore systems is evaluated.

- Chapter 7 includes a case study on how the POLARIS synthetic aperture radar application can be mapped to a multicore system on FPGA. The POLARIS SAR application requires real-time data processing for a 3000m wide area with a resolution of 2x2 meters. 64 Tinuso processor cores are required for this task. The case study shows that multicore systems on FPGA deliver a high computing performance at low power budget.
- Chapter 8 presents a proposal for a task based programming model and runtime system for Tinuso that support the programmer in expressing parallelism and efficiently executes parallel programs.
- Chapter 9 draws conclusions of the work presented in this thesis and discusses future work.
- Appendix A describes the Tinuso Application Binary Interface, *ABI*.
- Appendix B includes a detailed instruction reference.

Chapter 2

Terminology

In this chapter some key terms are defined that will be used throughout the rest of this thesis. These definitions might be different from those used in other works.

Microprocessors can be considered as the engine for the digital revolution. The steadily increasing performance of microprocessors has enabled innovations such as personal computers, Internet, smart-phones, and satellite navigation systems.

In the 1970s, the general trend in computer design was to implement more and more powerful instructions to make life easier for the programmers. It was a common thought that increasing the complexity of the instruction set architecture would best exploit the rapidly advancing semiconductor technology. One example of these complex instruction set computers, *CISC*, is the VAX 11-780, which included a total of 280 instructions [109]. However, compilers for high-level computer languages were not able to exploit complex instruction sets and rarely utilized these added instructions.

Patterson and Sequin therefore worked on a reduced instruction set computer, *RISC*, architecture [94]. They aimed to improve performance with a lightweight hardware design that can easily be optimized. The instruction set architecture of RISC-I comes with 31 instructions that all are of the same size and execute in a single cycle. Only load and store instructions access memory, while other instructions operate on a set of registers, called register file. Implemented instructions were selected with the perspective that compilers for high-level computer languages can make best use of it. The concepts of RISC got widely adopted in a large number of academic and commercial architectures

such as Alpha, ARM, MIPS, SPARC and PowerPC.

An important contributor for the success of RISC architectures was the ability to exploit the advancing semiconductor technology. Due to its simplicity, RISC architectures are easy to implement with an execution pipeline. Each pipeline stage operates on one instruction at a time. RISC pipelines therefore leverage instruction level parallelism by executing instructions partially overlapped. The performance of a RISC design can be improved by splitting up the time critical path into several combinatory - flip-flop sections, called pipelining. This leads to simple pipeline stages that enable a higher system clock frequency.

While a deep pipeline allows for a high system clock frequency, it incurs additional complexity in the control logic. For example, forwarding in the pipeline is a vital technique to limit the number of data-hazards by inherently moving results from a later pipeline stage to an earlier one. In a deep pipeline more pipeline stages need to be considered for forwarding. Therefore the forwarding logic may become large and complex and limit the system clock frequency. Instead of resolving hazards in hardware, it is possible to expose the hazards to software. While this reduces complexity of the hardware, the compiler has to be aware of the exact pipeline behavior and schedule instructions without hazards. Thus, this approach moves complexity from hardware to the compiler toolchain.

To improve performance, computer designers added multiple functional units to the pipeline to execute instructions in parallel. Very Long Instruction Word, *VLIW*, is an approach where multiple instructions are encoded in a single instruction word. These instructions then execute in parallel on different functional units. However, as there are many dependencies among operations, it is difficult to schedule instructions to utilize all functional units. This is a problem for instructions that change the control flow of a program, such as branch instructions. Due to data dependencies, most functional units remain idle until a branch instruction has completed, which makes branch instructions very costly in VLIW machines.

In computer architecture often branch prediction is used to reduce the costs of branch instructions. The pipeline does not wait until the branch direction is computed, instead branches as executed speculatively. If it is later detected that the prediction was wrong then the speculatively executed instructions have to be discarded and the pipeline restarts with the branch instruction, which reduces performance. Another possibility to reduce branch costs is leverage predicated execution to reduce the number of executed branch instructions [56]. Compilers for architectures with support for predicated execution aim to convert control dependencies that would usually implemented with a branch instruction into a data dependence that can be implemented with a predicated instruction.

Predicated instructions have an additional predication operand, called predicate. Depending on the value of the predicate, an instruction is executed or annulled and has no effect. Predicated instructions can therefore be used to implement simple if-statements without branch instructions. This technique is called if-conversion.

Modern microprocessors used in personal computers typically take a different approach to circumvent dependency problems than RISC processors. Instructions are fetched and register-renaming is performed before an instruction is placed in a ready queue. Dynamic scheduling and changing register names eliminate some data dependencies and allow for a parallel out-of-order execution on multiple functional units. Out-of-order and superscalar execution have proven to be very an effective techniques to extract parallelism at instruction level.

As the performance of the memory has not improved at the same rate as processor performance, the memory hierarchy has become a very important factor for the performance of computer systems. The memory hierarchy takes advantage of locality and cost-performance of memory technologies. The principle of locality states that most programs do not access all code or data uniformly, locality therefore occurs in time and space. The principle of locality and the fact that small memory structures can operate at a higher clock frequency led to memory hierarchies that combine memory of different speed and size. Each level of memory therefore maps addresses from a slower, larger memory to a smaller but faster memory higher in the hierarchy. The resulting memory system comes with cost per byte almost as low as the cheapest level of memory and a speed almost as fast as the fastest level [53].

However, computer designs have grown very complex and power hungry because of high clock frequencies. Modern high performance microprocessors have reached a point where the power density is so high that it is not possible to obtain more performance by increasing the clock frequency. Therefore, computer designers started to implement systems with multiple processing cores to increase the performance.

The history and evolution of microprocessors is highly relevant for this thesis as we faces similar design challenges when we evaluate architectural trade-offs to improve the performance of computing systems on FPGAs. FPGAs are reconfigurable devices that allow for implementing electronic circuits. The fabric of an FPGA typically consists of a two-dimensional matrix of configurable logic blocks, *CLBs*. A CLB contains small lookup tables, LUT, and successive configurable flip-flops as shown in Figure 2.1. Modern FPGA families typically come with 6-input lookup tables while previous generation FPGAs implement lookup tables with fewer inputs. Moreover, the fabric of modern FPGAs con-

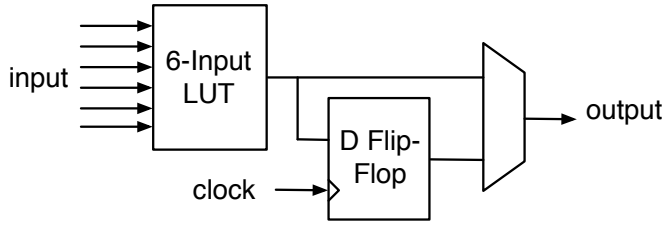
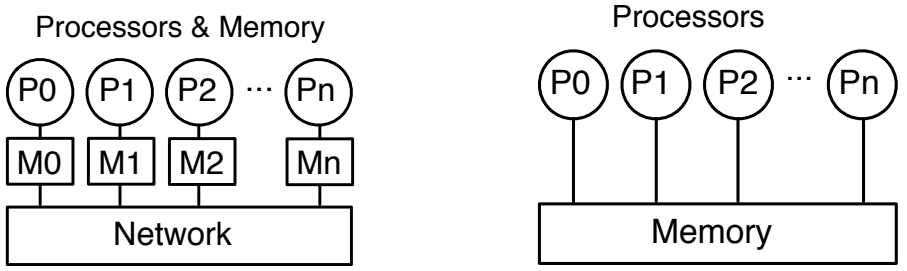


Figure 2.1: Simplified view on a configurable logic block

tains memory resources in the form of synchronous static SRAM blocks, *block RAMs*, scattered across the matrix. These block RAMs typically can store a few tens of kilobits of storage organized very flexibly. Designers rarely access FPGA resources directly. Instead, a *synthesis* tool is used to optimize and transform a logic described in a high-level behavioral description language to a low-level binary representation, which is used to configure all FPGA resources.

Embedded systems are computing systems that typically are designed for specific applications. Embedded applications may be described as electronic circuits and mapped to the FPGA or, alternatively, be implemented in software and execute on a synthesizable processor core. In this thesis we define a synthesizable processor core as a hardware description of a given processor architecture that can be synthesized for FPGA technology. The internal structure of FPGAs constrains hardware designs. Complex structures on an FPGA may easily become a critical path of the design and lower the performance. We define the critical path as the combinatory path with the longest delay. This combinatory path includes both interconnect delay and logic delay. We define interconnect delay as the propagation delays through the FPGA's interconnect and the programmable switches while logic delay are combinational delays in configurable logic blocks. The critical path limits the maximum clock frequency of a design. As complex multiported structures cannot be implemented efficiently on FPGA, we cannot exploit instruction level parallelism with superscalar out-of-order execution. Hence, the design of an instruction set architecture optimized for FPGA implementation addresses similar design challenges as RISC implementations in the 1980's.

As the computing power of a single processor pipeline is not sufficient for many applications, multicore systems are required to improve performance. Such systems include multiple processors that communicate through an interconnection network and have access to memory. There are two commonly used



(a) Message Passing (b) Shared Memory

Figure 2.2: High level abstraction of multicore systems

abstractions of multicore systems: message passing and shared memory systems.

Figure 2.2a illustrates a message passing system. Each processor core comes with a private memory. Processor cores communicate with explicit messages through an intercommunication network. Shared memory systems, on the other hand, implement a global shared memory as shown in Figure 2.2b. Communication between processor cores takes place implicitly via the shared memory.

In multicore systems, where processor cores maintain caches of a shared memory resource there may arise problems with inconsistent data. Once a processor core writes a memory location it is only updated in the local cache. Copies of this memory location that reside in caches of other processor cores are not updated automatically. Hence, data in the system may become inconsistent, which leads to a incorrect execution of a program. Thus, a mechanism is required, which updates memory locations in the system to keep memory consistent.

Coherency mechanisms assure that data written by one processor will eventually become visible to other processors and multiple writes to a particular location will be seen in the same order. However, coherence does not specify when writes will become visible to other processors. Therefore, a consistency model is required that defines the programmer's view of the ordering of memory operations to ensure a correct execution of a program.

In 1979, Lamport described a sequential consistency model to correctly execute parallel programs in multicore systems [69]. It requires write operations to become instantaneously visible globally to all processor cores. While this is very intuitive for the programmer it is very costly to implement [53]. Instead,

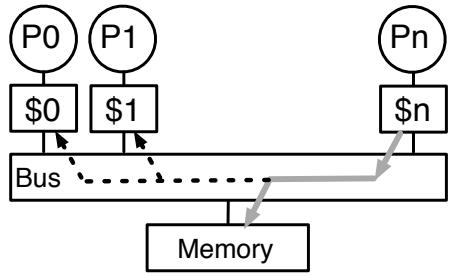


Figure 2.3: Snoopy cache coherent multicore system

relaxed consistency models have been developed that ease some restrictions of sequential consistency for most of the program, but enforce them in specific situations that are important for a correct execution of a program [36, 62]. This can for example be implemented with memory fences or barriers in the program code that require all previous memory accesses to complete before resuming the program execution.

Cache coherence protocols are designed to maintain the consistency between caches in multicore systems. A plethora of cache coherence protocols exist that can be classified as snoopy or directory based protocols.

Figure 2.3 illustrates the coherence mechanism in a snoopy bus based system. When a processor core writes data to main memory, the other processors in the system "snoop" on the bus and monitor transactions. If one of their cached memory blocks is involved in a bus transaction they update it. Snoopy protocols are easy to implement when the communication infrastructure is a shared medium such as a bus. However, in on-chip networks it becomes more difficult to implement snoop mechanisms and the scalability is limited.

Alternatively, directory-based cache coherency protocols maintain the state of memory blocks in a central directory. Figure 2.4 illustrates a multicore system with directory based cache coherence. A central directory includes a list of all processors that keep a copy of a cache block. Coherence is maintained as this directory is included in all memory transactions. However, directory-based cache coherency protocols are very complex and difficult to scale [5]. Moreover, cache coherency may incur a large hardware overhead and significantly increase the power consumption of a system. Hence, it is an active area of research to improve the scalability and power efficiency of cache coherence protocols [13, 19, 86, 123, 124].

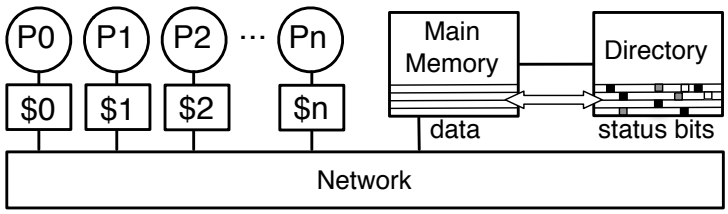


Figure 2.4: Directory based cache coherent multicore system

Beside memory consistency, parallel programming is a major challenge for multicore systems. Parallel programming has been studied for decades for high performance general-purpose computer systems and it is still an active ongoing area of research. Researchers aim for programming languages, programming models, and runtime systems that support the programmer expressing parallelism to exploit parallel hardware architectures.

A programming model bridges the gap between the underlying hardware and the application software. The programming model is an abstraction of the underlying computer system that provides certain operations to the programmer. In parallel computing, the programming model aim to support a programmer in expressing parallelism and to enable an efficient mapping of applications to hardware.

The runtime system includes hardware and software resources that enable the execution of a program on a computer system. A runtime system implements a broad range of functions depending on the programming language. For example, main objectives of the c runtime are to insert instructions to manage the processor stack, access memory, and processor interfacing. Additional functionality a runtime may implement is type checking, code generation, debugging, and optimization services. In addition, the runtime of parallel programming languages manages resources such as tasks and communication on behalf of the application.

A commonly used abstraction level is to consider parallel applications as a set of tasks. We define a task as an independent part of a program. Parallel programs can be divided into a set of tasks that then may execute on different processor cores.

Chapter 3

Processor Core

As FPGAs get more competitive, synthesizable processor cores become an attractive choice for embedded computing. The advantage of synthesizable processors cores is that they can easily adjust to the needs of an application. The ability to reconfigure FPGAs allows for adapting the design at any time, also when the embedded system is already in operation. Moreover, FPGA designs that include synthesizable processors tend to have a lower hardware complexity and a lower system cost compared to hardware designs that consist of a discrete microprocessor and an FPGA device.

However, synthesizable processor cores typically have a lower performance than discrete microprocessors. This chapter evaluates performance aspects of synthesizable processor core and examines their impact in the design of new processor architecture, Tinuso. Tinuso is a processor architecture optimized for performance when implemented on an FPGA.

This chapter is based on two publications [103, 105] and is structured in the following sections. Section 3.1 motivates the basic ideas of the Tinuso architecture. Section 3.2 discusses related work and highlights other synthesizable processor cores. Section 3.3 gives an overview of the Tinuso architecture. The instruction set architecture is described in Section 3.4 while a detailed description of all instructions can be found in the Appendix B. Section 3.6 describes the hardware implementation and presents processor internals such as forwarding mechanism, predication support, and cache implementation. In section 3.7 the processor core is evaluated by comparing clock speed, hardware resources, and code execution time with commercial synthesizable processor cores. Section 3.8 summarizes and concludes this chapter.

3.1 Introduction

As the power efficiency and logic capacity of FPGAs increases, they become an attractive technology for use in low and medium volume markets. For example, Xilinx's Virtex 7 family comes with devices up to two million logic cells [118]. Their power consumption is 50% lower than previous generation FPGAs. These devices allow for combining multiple processor cores, dedicated accelerators, and a large number of interfaces on a single device.

Hardware designers often use synthesizable components to compose systems. Synthesizable components are descriptions of logic circuits that are converted, also called *synthesized*, to a low-level binary representation to configure the FPGA fabric. There exists a broad range of synthesizable components including processor cores, arithmetic units, signal and data processing blocks, and all sorts of interfaces.

Nevertheless, synthesizable processor cores have a significantly lower performance than discrete microprocessors because of two reasons. First, the logic circuit of discrete microprocessors is directly mapped to the chip, which enables an efficient implementation of complex hardware structures. Synthesized designs on FPGAs, on the other hand, are limited to use the predefined reconfigurable resources. Second, commercial synthesizable processor cores are primarily designed to be highly configurable than to deliver a highest possible performance. For example, neither Xilinx MicroBlaze nor Altera Nios II fully exploit current FPGA architectures as they do not make use of pipelined memory resources [8, 120]. Open-source processors tend to have a low performance as they are not optimized for any specific target technology. Most synthesizable processors available today implement a RISC like pipeline with 3 to 6 pipeline stages [93].

We research opportunities to exploit current FPGA architectures that improve the performance of synthesizable processor cores. Thus, we have to apply a technique to exploit instruction level parallelism that maps well to the internal structures of FPGAs. Out-of-order and superscalar execution have proven to be very effective techniques to extract parallelism. Multiple instructions execute in parallel and dynamic scheduling with register renaming effectively reduces data hazards. However, this comes at the cost of complex multiported structures that do not perform well on FPGA architectures [42, 89].

Very long instruction word, *VLIW*, refers to a processor architecture where multiple instructions are executed in parallel on several execution units. In contrast to superscalar machines, instructions are scheduled statically. Data hazards are ether eliminated by the compiler or the hardware has to stall until

data dependencies are resolved, which lowers the performance. Although VLIW machines tend to be less complex than superscalar architectures the register file remains the bottleneck of the design as it requires multiple write ports if multiple instructions shall complete simultaneously. Purnaprajna and Ienne proposed to introduce an embedded multiported RAM for register file implementation in future FPGAs to mitigate this performance bottleneck [97]. Instead, we focus on single in-order pipelines as they provide the best energy-performance trade-off for processors of up to 300 million instructions per second [11].

FPGAs typically consist of an array of configurable logic blocks that contains small lookup tables and successive configurable flip-flops. Thus, pipelining, i.e., splitting up the critical path into several combinatorial - flip-flop sections, is relatively inexpensive in FPGAs [39]. Moreover, modern FPGAs come with pipelined memory blocks that allow for implementing fast caches and register file.

Superpipelining breaks pipeline stages into smaller stages and leverage instruction level parallelism by executing operations partially overlapped. Shorter pipeline stages reduce the clock period and thus allow for a higher clock rate and instruction throughput [60]. Superpipelining may lead to a higher system clock frequency but adding pipeline stages breaks the compatibility to existing instruction set architectures. Hence, compilers and operating systems need to be adapted and all existing software need to be recompiled. Moreover, adding more pipeline stages increases the hardware costs and complicates the control logic of a processor core, which may be a limiting factor. We therefore research if superpipelining is a viable technique to improve the performance of synthesizable processor cores.

Designing an instruction set architecture is therefore connected with a large workload but it opens up possibilities to remove legacy artifacts and to optimize the hardware implementation. For example, the performance of a superpipelined processor heavily depends on how efficient branch instructions are implemented. Branch instructions become expensive as branch address and direction are resolved late in the pipeline. For processors with pipelined caches and register file there are at least four successive instructions fetched until the branch address is computed. These instructions are called *delay slots* that the compiler has to attempt to fill with useful independent instructions. If unsuccessful, the compiler will have to insert no-operation, *nop*, instructions which will reduce performance.

One possibility to reduce branch costs is to leverage predicated instructions. A predicated instruction is executed if a condition that is specified in the operation code is true, otherwise the instruction is annulled and has no

effect. It enables compilers to replace some conditional control-flow sequences with predicated instructions and hence reduces the frequency of branch instructions. Moreover, predicated instructions can be used for filling branch delay slots with instructions from before the branch, from the branch target, and from fall through.

A processor pipeline requires control logic to prevent situations where hardware would produce wrong results, called hazards. For example, multiple instructions may attempt to use the same hardware resources or an instruction may want to use data before it is available. Adding pipeline stages to a processor pipeline design increases the complexity of this control logic, which may utilize a large number of hardware resources and thus become a limiting factor for the system clock frequency. We therefore expose hazards to the compiler, which has to schedule instructions that the processor pipeline can execute them without stalling. Thus, we obtain a very lightweight design but need to deal with additional complexity in the compiler.

We examine the impact in the mentioned performance aspects in the design of new processor architecture, Tinuso. Tinuso is a statically-scheduled, single-issue, RISC processor architecture that uses a deep pipeline to attain a high system clock frequency. Predicated instructions help to prevent costly pipeline stalls due to branches. By letting the compiler take care of all types of hazards we obtain a lightweight hardware design.

3.2 Related Work

Major FPGA vendors such as Xilinx, Altera and Lattice Semiconductors offer processor cores optimized for their technology. These cores are highly configurable and come with a large number of peripherals and rich tool-chain support. Xilinx MicroBlaze and Altera Nios II come as optimized netlists of vendor-specific primitives. Hence, they are bound to the vendor's hardware and tool-chains. Lattice Semiconductors LatticeMico32 core is licensed under an open intellectual property core license. It is available in synthesizable register transfer language and can be ported to any FPGA family.

Xilinx's MicroBlaze is a popularly used synthesizable processor that implements a 32-bit Harvard RISC architecture with a rich instruction set optimized for embedded applications. A branch target buffer provides dynamic branch prediction. Xilinx's Embedded Development Kit, *EDK*, is used to configure MicroBlaze. A large amount of peripherals, memory and interface features are available to adapt the processor to a given application. MicroBlaze utilizes an

in-order single issue pipeline with a load-store instruction set. The performance optimized MicroBlaze configuration utilizes a five-stage pipeline.

Altera's current equivalent to MicroBlaze is called Nios II [9]. The Nios II family includes three processors that are optimized for highest performance, smallest size, and performance and size balanced implementation. In contrast to MicroBlaze, Nios II processors are not bound to a specific technology. Beside FPGAs, they can be implemented on Altera's HardCopy specific integrated circuits, *SICs*. NIOS II allow for up to 256 custom instructions.

Ehliar et al. [4] introduce a processor architecture optimized for FPGA implementation. They implement a deep pipeline with limited register forwarding to provide a high system clock frequency. DSP functionality in terms of multiply-and-accumulate is built-in. The processor can be clocked at a frequency as high as 357 MHz on a Xilinx Virtex 4, FPGA. Lack of cache resources, limited register file entries, and limited address spaces make this architecture work well for DSP-like applications, but prevent their adoption for large-scale multicore platforms.

There are a number of synthesizable processor cores available as open-source designs. Examples are LEON3 and OpenRISC 1200. They are not optimized for any specific target technology and therefore have sub-optimal performance when implemented on an FPGA [112]. OpenFire is an open-source, binary-compatible MicroBlaze clone written in Verilog. OpenFire was designed with the aim to have the entire HDL sources available for research in embedded multicore architectures [32]. OpenFire attains approximately the same clock frequency as MicroBlaze. OpenFire comes with a limited system bus and therefore requires fewer hardware resources than MicroBlaze.

The LatticeMico32 is another open-source processor design provided by Lattice Semiconductors. It is available in synthesizable register transfer language and can be ported to any FPGA family. LatticMico32 is an in-order single issue processor with a load-store instruction set. Static branch prediction is supported only. The pipeline has six stages. Mico System Builder, *MSB*, implements the LatticeMico32 processor and attaches peripheral components. LatticeMico32 is optimized for FPGA implementation and provides the highest clock frequency among open-source soft-cores.

The mentioned synthesizable processor cores come with 3 to 6 stage pipelines. All utilize a single execution stage only, which keeps the forwarding logic simple. While the LM32 implements static branch prediction both MicroBlaze and Nios II leverage a branch history table to reduce the branch penalty. These pipelines are able to stall if data dependencies on instructions with delay slots, such as loads or shifts, are detected. Compilers have to fill at most one branch

delay slot and do not need to worry about data hazards when scheduling instructions. Moreover, none of these architectures supports predicated instructions, which keeps compiler backends simple.

While there are a plethora of processor architectures that leverage predicated execution, ARM is the only synthesizable processor architecture that comes with predicated instructions. ARM processors have an instruction set where the majority of all instructions are predicated. Instructions are executed depending on the value of processor flags. Instructions such as compare or subtract can set the processor flags. ARM provides the ARM Cortex-M1 core, a synthesizable processor designed specifically for FPGA implementation. It can be clocked at frequencies as high as 200MHz on Virtex 5 and Stratix III devices [10]. Tinuso's predication scheme differs from ARM architectures as predicated instructions are executed depending on the value of predicate registers. Thus, any arithmetic or logical instruction can set these registers and its value can be preserved for a longer period.

Texas Instruments c6x is a digital signal processor platform that supports predication execution of all instructions [56]. In contrast to Tinuso, c6x is a very long instruction word, *VLIW*, machine with eight parallel function units. As c6x comes with 5 branch delay slots, there may be up to 40 instructions issued before a branch takes place. Often, control and data dependencies limit the number of independent instructions that can be used to fill these delay slots. As a consequence c6x leverages predicated instructions to replace conditional control flow instructions.

3.3 The Architecture of Tinuso

Tinuso is a statically scheduled, single-issue, RISC processor architecture optimized for high throughput when implemented in FPGAs. Register file size, cache sizes and cache organization are chosen to match the memory resources found in modern FPGAs. Tinuso applies superpipelining to exploit instruction level parallelism. Superpipelining breaks pipeline stages into smaller stages to reduce the instruction cycle time. However, superpipelining cannot be applied arbitrarily, with increasing pipeline depth, the clock frequency can be increased until a point where the control hardware gets too complex putting a lower bound on the delay along the critical path.

To keep the implementation simple, Tinuso executes instructions in-order and does not support speculation beyond delayed branches. Only a single instruction is issued at a time. To further simplify the implementation, there is

only limited support for interrupts and exceptions. The pipeline, with side-effects, is fully exposed to software. Thus, the software will have to consider all types of hazards. Register forwarding in the pipeline is a vital technique to limit the number of data-hazards by inherently moving results from a later pipeline stage to an earlier one. Missing forwarding logic in a deep integer pipeline, similar to Tinuso, causes a performance slowdown of almost a factor of two [7]. Consequently, Tinuso supports full forwarding. Thus, forwarding is possible from all pipeline stages. Tinuso is a three-operand architecture with a fixed instruction word length of 32-bits and a large register file with 128 entries. Hence, there is limited space for instruction encoding. To keep the number of instructions low, the integer pipeline only supports data types such as signed and unsigned 32-bit words. The reduced instruction set is designed to support C and assembly language programming.

Tinuso leverages a single register file that is used for integer and floating point operations as well as for predicates. A floating point unit may operate on double precision operands, meaning that two 32bit registers are used to form one 64bit operand. As a consequence, it takes two clock cycles to load and store floating point operands. Tinuso comes with optional functionality that can be configured when the design is synthesized. Currently, optional functionality includes a multiplier, barrel shift operations, floating point unit, co-processor interface and communication infrastructure.

3.3.1 Memory Hierarchy

Hierarchically structured memory is a hardware optimization that exploits the principle of locality. The memory hierarchy includes a large main memory and small memory blocks, called caches that are tightly integrated with the pipeline. Caches store copies of frequently used memory blocks with the aim to reduce the average time to access memory. Hence, if mostly cached memory locations are accessed, the average memory latency will be closer to the cache latency than to the latency of main memory. As the access latency to main memory typically is high, the memory hierarchy is an important performance factor of a computer system. Modern FPGAs contain memory resources in the form of synchronous static RAM blocks, *block RAMs*. These block RAMs can typically store 9-36 kilobits of data organized very flexibly. Data can normally be accessed via two memory ports and the data word width of each port is configurable. Current state-of-the-art FPGAs can have up to hundreds of these block RAMs. The Tinuso processor architecture is optimized for FPGA implementation by balancing speed and hardware resources. Hence register file, instruction

and data caches are implemented as block RAMs. Clocking the pipeline at a frequency that is close to the maximum clock frequency of block RAMs requires a pipelined implementation. As a consequence, a pipelined cache or register file access takes two clock cycles. This has a major impact on the processor architecture. The consequences for instruction scheduling, cache miss handling, and the implementation of branch instructions are described within this chapter.

3.3.2 Predicated Execution

Superpipelined processors resolve branch address and direction late in the pipeline. Hence, branch instructions complete with a delay of multiple cycles. In a processor design with pipelined instruction cache and register file at least four successive instructions are fetched until the branch completes. These instructions are called delay slots that a compiler attempts to fill with useful independent instructions. If unsuccessful, the compiler will have to insert no-operation instructions, which will reduce performance. Therefore, branch instructions are expensive in superpipelined processor cores.

Consequently, Tinuso leverages predicated instructions to reduce the number of dynamically executed branch instructions and to fill delay slots. Predicated instructions have an additional predication operand, called predicate. Depending on the value of the predicate instruction is executed or annulled and has no effect. Tinuso holds predicates in predication registers, which are a subset of the register file. In total, four bits out of the instruction word are used to encode predicated execution. Three bits are required to address the eight predication registers. An additional negation bit N can be used to negate the predication condition of an instruction. The eight predication registers are Boolean representations of the lowest eight registers in the register file. Thus, predication registers are set by any instruction that writes to the lowest eight registers in the register file.

In computer architecture, branch delay slots are the pipeline slots that come after a branch instruction, but before the branch target address has been computed. Instructions in branch delay slots are executed regardless of whether the branch is taken or not. Compilers try to fill those slots with instructions that would be executed before the branch if there were no delay slots. Since it is difficult to fill all branch delay slots with useful instructions often `nop` instructions need to be inserted. Tinuso's predication scheme allow for filling branch delay slots with instructions from before the branch, from the branch target, and from fall through. In addition, the use of predicated instructions allows a compiler to replace some conditional control-flow sequences with predicated instructions

and hence reduces the amount of branch instructions and thereby lower the total amount of branch delay slots of a program. Predicated instructions can also be used to selectively annul instructions in the branch delay slots.

The Tinuso architecture provides predicated execution for of all instructions except of the instruction that moves immediate constants into a register. This `move high` instruction is not predicated because the large immediate takes up to much space in the instruction word that there is no space left to encode the predicate. Instead, a `move high` to a temporary register and a successive `add immediate` instruction can be used to conditionally load 32 bit constants into a register. Given the relatively large register file, there is limited use of making `move high` instructions predicated.

3.4 Instruction Set Architecture

Tinuso implements a load-store instruction set with a fixed 32-bit instruction word length. Given the large amount of memory resources in modern FPGA's, a large register file with 128 entries was selected. Thus, seven bits are required to encode an operand. Moreover, predication takes up four bits of the instruction word encoding. Another three bits are used to encode instruction types. Hence there is limited space for instruction encoding. For that reason only instructions strictly required to support C and assembly language programming have been implemented.

Figure 3.1 illustrates the encoding of instruction types. There are up to two source operands `rA`,`rB` and one destination operand `rC`. The `N` represents the negation bit for predicated execution.

Tinuso uses three bits of the instruction word to categorize instructions in eight instruction types.

One instruction type is used for arithmetic and `compare-and-set` instructions. Four bits are used to encode the functionality of 15 three-operand instructions. The 16th value is used to encode the subtype of arithmetic instructions with two or one operands. Three operand operations include addition, subtraction and logic operations. Tinuso supports three `compare-and-set` instructions that set a destination register according to the comparison of the two input operands. In contrast to the MIPS architecture, where five bits out of the instruction word are reserved for defining the shift amount, Tinuso supports a small set of commonly used shift instructions only.

Tinuso only supports memory accesses in 32-bit word length. Loading and storing byte or half-word data types is accomplished with shift and mask oper-

ations. Eleven bits are reserved for offsets, allowing to address memory relative in the range of +/- four kilo bits.

An immediate can be considered as a constant value that is part of an instruction, commonly used to manipulate a register content. To move immediate values into registers the `move high` instruction is used. It loads a 22-bit immediate into the upper part of a register. Hence a successional `add immediate` instruction is required to load a 32bit constant. The 22-bit immediate of the `move high` instruction is split up to simplify the instruction decoding logic hardware.

Tinuso provides branches relative to the program counter. There are two branch instructions: one that branches if the content of the input operand is zero and one that branches if it is not zero. Branch instructions come with a 15-bit offset, which is shifted left two bits, sign extended, and added to the program counter. Tinuso branch instructions allow for PC relative jumps in the range of +/- 16 kilo bytes.

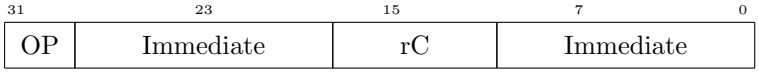
The instruction set also includes a `jump register` instruction. Absolute jumps can be composed by a `move high`, an `add immediate` and a `jump register` instruction. Thus, absolute jumps become expensive. Still, we consider this to be an appropriate compromise given the fact that absolute jumps represent less than 1% of all executed instructions in typical benchmark applications [53]. To support function calls, the `jump register` instruction has built-in linking functionality, which stores the return address in a dedicated link register.

In addition, there is some space left in the ISA to encode optional functionality such as multiplier, barrel shift operations, floating point unit, co-processor interface, and communication infrastructure and additional instructions. A detailed description of all instructions can be found in the Appendix B.

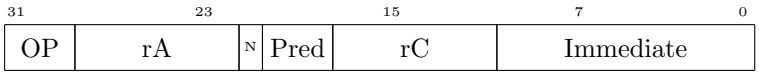
3.5 Pipeline Architecture

The current design consists of eight pipeline stages as shown in Figure 3.2. The program counter is not considered as a separate pipeline stage. The Tinuso processor architecture makes use of block RAMs for caches and register file. To attain a highest possible clock frequency, the block RAM access is pipelined by adding optional output registers to the block RAM. As a consequence, a cache or register file access in the Tinuso processor architecture takes two clock cycles. Hence, the instruction and the corresponding cache tag are not accessible until the instruction decode stage. Provided there is a tag match, the instruction is decoded whereas the pipeline invalidates the instruction on a cache miss. When

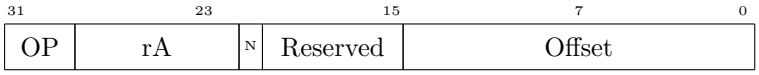
- Move High Instruction:



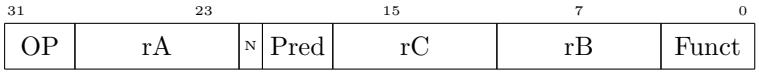
- Add-Immediate Instruction:



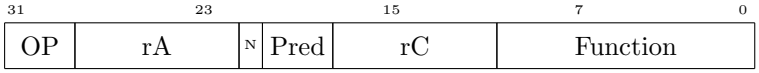
- Branch Instructions:



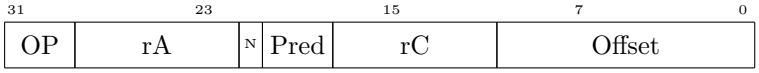
- Arithmetic, Compare-and-Set Instructions with three Operands:



- Arithmetic Instructions with two Operands:



- Load Instruction:



- Store Instruction:

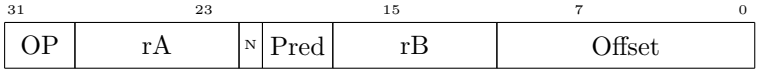


Figure 3.1: Overview of Tinuso's instruction word encoding

the cache miss is resolved, the program counter is set back to the value it held before the cache miss occurred.
In the register fetch stage, the register file is accessed and the first portion of forwarding is performed.

Tinuso implements a unified register file with 128 entries. A single register file is used for integer and floating point operations and predicated execution. Hence, the register file has two ordinary read ports that can access any of the 128 entries and one simplified read port to access the eight predication registers.

To let the processor take advantage of fast cache access and register file access it is necessary to pipeline the execution stage. It is currently split up into two stages. In the execution stages there is an arithmetic path and a logic path where instructions are decomposed into two sub-operations. In a deep pipeline, register forwarding is vital factor for the performance of the processor. Consequently, the Tinuso architecture supports forwarding from all pipeline stages.

To compute a branch target address, it is necessary to access the register file and determine whether a given register value equals zero. According to Figure 3.2, this cannot be done until the first execution stage. Hence, there are four branch delay slots.

The data cache is accessed in the memory stage. Again this takes two clock cycles, meaning that when a cache miss is detected, there might already be a new instruction in the input register of the block RAM. Similarly to the instruction cache, this is handled by flushing the pipeline when a data cache miss occurs and restarting the pipeline once the cache miss is resolved. In the multiplexer stage the data cache tag check is accomplished and results of the execution stages and load instructions are multiplexed. Finally, results are written back to register file in the write-back stage. There is a single write port that is used by all instructions to write results back to the register file.

In Tinuso, the cache controller is able to control the pipeline by setting the program counter. This allows for flushing the pipeline on cache misses and to restart after the miss has been resolved. Moreover it is able to halt the pipeline on external request. In multicore systems, this is an important feature to implement a cache coherency protocol in hardware.

3.6 Hardware Implementation

The current VHDL implementation, Tinuso I, is a full implementation of the Tinuso architecture. It is designed explore the feasibility of the architecture

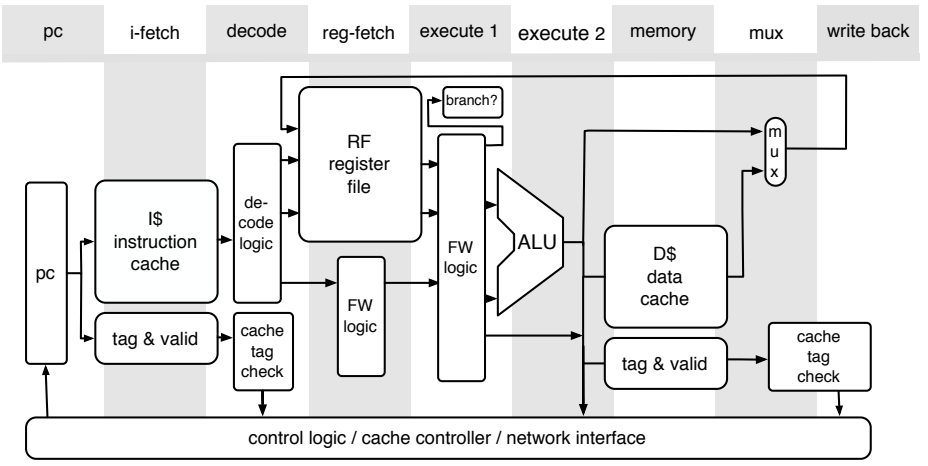


Figure 3.2: Pipeline sketch of an 8 stage Tinuso implementation

in terms of speed and required hardware resources. The hardware prototype includes first level cache controller and a memory interface. Tinuso I's eight stage pipeline is shown in Figure 3.2. Pipelined memory resources are provided by current state-of-the-art FPGAs. Hence, register file and caches are implemented as block RAMs with optional output registers. To let the processor take advantage of fast cache and register file accesses the execution stage is pipelined into two stages.

3.6.1 Register Forwarding

Register forwarding in a deeply pipelined processor pipeline is vital for its performance. However, it is a challenge to implement register forwarding in a deep pipeline without limiting the maximum clock frequency of the system. Tinuso requires forwarding from a total of six stages. Typically, a large multiplexer, placed in the execution stage, selects among results from successive pipeline stages and register file.

Current FPGAs typically utilize six-input LUTs that can be configured to implement a simple 4:1 multiplexer. Multiplexers with more inputs require a cascaded implementation. For example, in Xilinx Virtex 5 FPGAs, 8:1 multiplexers are composed by a dedicated two-input multiplexer (F7AMUX / F7BMUX) that selects among the outputs of two LUT based 4:1 multiplexer [122]. Cascading

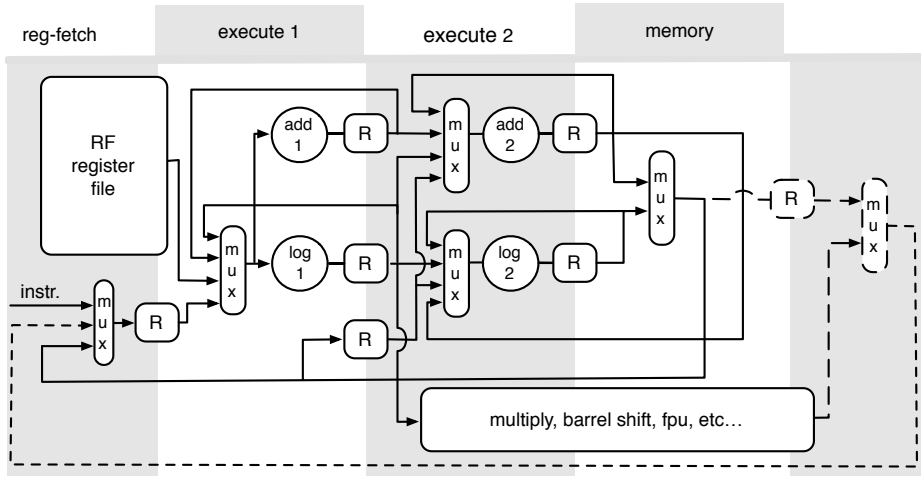


Figure 3.3: The pipelined forwarding mechanism in Tinuso

multiplexers increases the propagation delay through logic gates. Moreover, the interconnect delay for all multiplexer inputs need to be taken into account when doing a timing analysis of the design. Considering a cascaded multiplexer and the interconnect delay for all forwarding paths, this forwarding logic becomes a limiting factor for the system's clock frequency. Consequently, Tinuso leverages a pipelined forwarding logic. Tinuso's approach for a fast forwarding logic in the execution stages is twofold. First, only forwarding from adjacent pipeline stages is permitted to limit the interconnect delays. Second, forwarding multiplexers in the critical path of the design are limited to a single four-input instance to keep the levels of logic low.

The proposed solution is illustrated in Figure 3.3. In the register fetch stage, there is a large multiplexer that selects among results from most pipeline stages. One input of this large multiplexer is connected to the instruction decode stage, this is necessary to load immediate constants into the execution stage. This multiplexer combines several forwarding paths into a single register. A priority encoder makes sure that this register always carries the value of the most relevant forwarding source.

There is an arithmetic, *add*, execution path that implements instructions such as addition, add immediate, load and store. Logic instructions are computed in a dedicated logic, *log*, execution path. The arithmetic path is time

critical. Experimental results have shown that we attain the highest clock frequency when 20 bits of the 32-bit data word are computed in the first execution stage. Logic operations are pipelined in the same way to provide forwarding from the arithmetic path to the logic path and vice versa. While this approach increases the number of forwarding sources, it reduces the number of multiplexers in the time critical path.

The operands in the first execution stage can either be taken from the register file directly, from the result of the arithmetic or logic path, or from the register that carries the value of the remaining forwarding sources. Thus, there is only one four-input multiplexer in the first execution stage that handles forwarding. The same approach applies to the second execution stage. Again, the operand data can either be taken from the output of the first execution stage, from the result of the arithmetic or logic path or from the result of the memory stage. Tinuso's register forwarding implementation can be characterized by:

- Forwarding from all pipeline stages is possible.
- The forwarding multiplexers in the execution stage connect to adjacent pipeline stages only to keep routing delays low.
- In the time critical path of the execution stages, forwarding is implemented with a single four-input multiplexer only.

Optional instructions such as barrel shifts, integer multiplication operations, or floating point operation only partially use the pipelined ALU and implement a separated execution path. While it is always possible to forward results to these instructions, there is limited forwarding in the opposite direction only.

Figure 3.4 illustrates an example of Tinuso's forwarding mechanism in the pipelined ALU. In this example, there is an `add` instruction followed by an `or` instruction. The result of the addition is used as an operand of the `or` instruction. As these instructions are scheduled right after each other we need to forward the result from one instruction to the other. Figure 3.4-a shows that the addition uses the arithmetic execution path of the first execution stage. After one clock cycle the `or` instruction enters the logic execution path in the first execution stage while the `add` instruction is completed in the second execution stage. As shown in Figure 3.4-b the first part of the result of the addition is forwarded to the logic execution path. In a next step the `or` instruction has moved to the logic path in the second execution stage as shown in Figure 3.4-c. The preceding `add` instruction has now completed and the second part of the result is forwarded to the logic execution path in the second execution stage.

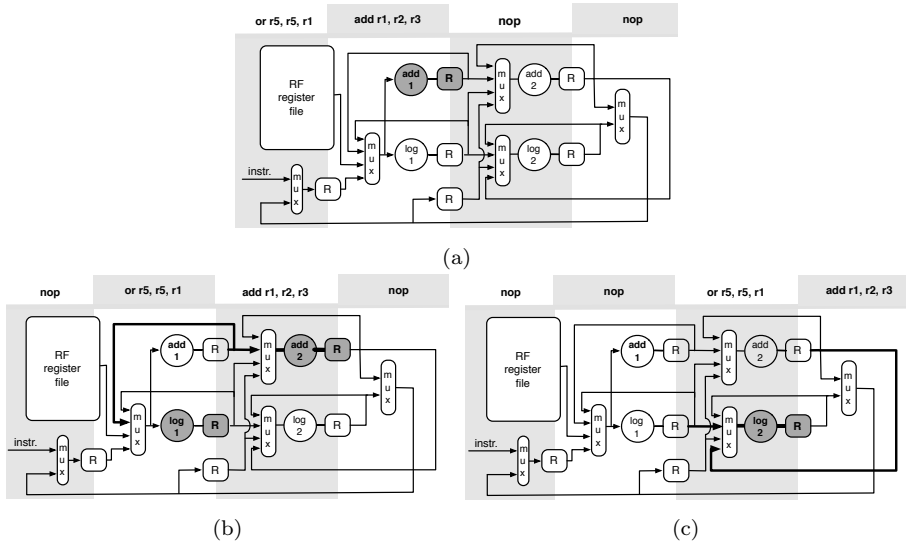


Figure 3.4: Example of pipelined forwarding

3.6.2 Pipeline Anomalies

Tinuso's forwarding mechanism is complex because there are many intermediate results that need to be forwarded without that the forwarding logic becomes too complex and limits the maximum clock frequency of the system. Moreover, also predicate operands need to be considered to avoid forwarding from annulled instructions. However, there are situations where forwarding in hardware cannot be implemented efficiently. It is therefore up to the compiler to avoid such situation.

Listing 3.1 shows a situation where forwarding from a predicated instruction goes wrong. The difficulty of forwarding from predicated execution is that there are situation where we do not know whether the predicated instruction is annulled or not until the second execution stage. This is the case if the result of the first instruction is used as predicate for instruction. The problem is that the third instruction is at that point in the execution stage and forwarding multiplexers are set already. If the result of the first instruction is 0, we forward the annulled result of second instruction to the third instruction, which results in a data hazard. Therefore Tinuso requires the compiler to add a nop before or after the predicated instruction to avoid a data hazard.

Listing 3.1: A Data hazard can be caused by forwarding the result of a predicated instruction from the second to the first execution stage

```
1 /* add r2 and r3 and save result in r1*/
2 add  r1, r2, r3
3 /*if r1 is not zero: add r1 and 5 and save result
      in r22*/
4 [r1] addi r22, r1, 5
5 /*add r22 and r1 and save result in r4*/
6 /*data hazard if instruction 2 is annulled!*/
7 add  r4, r22, r1
```

Listing 3.2: There is no data hazard in this situation because the value of register 1 is known when selecting the forwarding sources for the instruction on line 7

```
1 /* add r2 and r3 and save result in r1 */
2 add  r1, r2, r3
3 /* if r1 is not zero: add r1 and 5 and save result
      in r22 */
4 [r1] addi r22, r1, 5
5 [!r1] addi r22, r1, 2
6 /* add r22 and r1 and save result in r4 */
7 add  r4, r22, r1
```

However, Tinuso's forwarding logic supports situations as described in Listing 3.1. In this example the second instruction and third instruction use the same predicate but come with a different negate bit. Hence, only one of the instructions will be executed. The result of the first instruction determines which one will be annulled. This is known before the fourth instruction enters the execution stage. Thus, the forwarding multiplexers are set correctly.

Compare-and-set instructions also require careful scheduling to avoid hazards. Compare-and-set instructions utilize the arithmetic path in the execution stages where the second operand is subtracted from the first operand. While this subtraction is performed a number of status bits are set. These status bits are kept inside the pipeline and are not accessible for the programmer. Instead they are used to determine whether the instruction will set the destination reg-

ister to true or false. This is done in the memory stage. The buffered result of the memory stage is used to forward results from compare-and-set instructions, which therefore come with a total of three delay slots. Compare-and-set instructions are often used in combination with a successive branch instruction. The hardware is optimized for this instruction sequences and partially computes the result of the compare-and-set instructions in the second execution stage. The control logic of the program counter multiplexer then does the final computation to determine whether the branch is taken or not. Hence, there is only one delay slot required.

Load instructions compute the memory address in the execution stages. This address then points out which element of the cache to select. It takes two clock cycles to load data from the cache, one cycle to analyze the cache tag, and one cycle to buffer the forwarding result. Hence, load instructions have a total for four delay slots.

Write instructions compute the memory address in the execution stage. The memory address is used to load the corresponding cache tag. In the multiplexer stage the tag is checked for validity. If there is valid data and the cache tag fits, data is written to the cache during the write back stage. If a successive load instruction attempts to access the same memory location as the store instruction that has not completed yet, a read after write hazard occurs. Tinuso implements the data cache as block ram with a read first policy, as it enables the fastest implementation [121]. However, this comes at the cost of three instructions that need to be placed between a store and a load to the same memory location.

Other instructions, such as barrel shift, integer multiplier and floating point operations that only partially use the logic or arithmetic path of the execution stage have delay slots as well. However, the number of delay slots depends on the implementation and it is a trade-off between hardware costs and complexity on one side and performance and compiler complexity on the other side.

3.6.3 First Level Caches

Tinuso implements a hierarchical memory hierarchy. Instruction and data caches are implemented as block RAMs. For simplicity, a directly mapped cache organization was chosen. Directly mapped caches are very fast and can be directly implemented using the block RAMs. However, they also suffer from a relatively large number of collision misses. We believe this to be a fair decision given the actual cache size and the architectural goal to avoid branches by making use of predicated instructions. The block RAM size in the Xilinx Virtex 5 and newer FPGA families is 36 kilo bits. These memory blocks can be addressed as

two independent 18 kilo bit block RAMs. Tinuso currently implements cache size of 4 kilo bytes to ensure that the cache-data fits into a single block RAM. Typically these block RAMs are placed in a row on the chip, which allows for implementing a fast cache consisting of multiple block RAMs. Cache sizes and cache line sizes can be configured when the design is synthesized. The data cache uses a write back strategy to reduce the memory bandwidth, which is relevant for multicore platforms.

3.6.4 Cache Controller

The Tinuso hardware prototype includes first level cache controller and a simple memory interface. The cache controller includes two finite state machines, *FSM*. The first FSM is tightly integrated in the pipeline to detect cache misses and to flush the pipeline. The second FSM then translates cache misses into memory requests and is responsible for updating the caches. Figure 3.5 illustrates the state diagram of the cache controller FSM for the pipeline. The cache controller FSM remains in the idle state until a cache miss is detected. On a cache miss, the current program counter is saved and the pipeline is flushed. Then the control is then passed over to the cache controller FSM outside the pipeline, which requests the missed memory address and writes memory addresses back to main memory if required. Once the cache controller has received the requested memory data it updates the cache and passes the control back to the cache controller FSM in the pipeline to restart the pipeline.

Instruction cache misses are detected early in the pipeline. Hence, it is possible that there are control flow instructions, such as branches or jumps, in the pipeline that have not yet completed when an instruction cache miss occurs. The cache controller detects these branch and jump instructions and stores its program counter instead of the program counter of the instruction that caused the instruction cache miss. Once data is available in the instruction cache, the pipeline restarts with the program counter of the branch and jump instruction to ensure the correct control flow of the program. Data cache misses are detected late in the pipeline. Hence there are no outstanding control flow instructions to detect. Data cache misses are resolved with a higher priority than instruction cache misses. The advantage of this simple design is that it only needs to store one program counter value. Nevertheless, there can only be one outstanding cache miss at a time. Flushing and restarting the pipeline causes additional delay cycles compared with an implementation that is able to stall the pipeline. However, the hardware is simpler and allows for a higher system clock frequency.

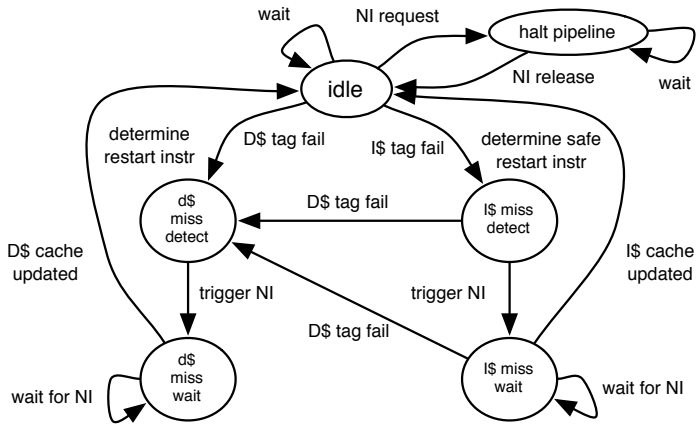


Figure 3.5: State diagram of the Tinuso cache controller

The cache controller is designed with the perspective that the pipeline can be halted on a network request. Once the pipeline is halted the cache controller may update the caches according to the network request.

3.7 Results

This section contains an evaluation of the proposed processor architecture where the performance of Tinuso I is compared with current state-of-the-art synthesizable processors. First, the maximum clock frequency and the required hardware resources for various FPGA families are derived and compared to a similar MicroBlaze configuration. Second, Tinuso I's branch performance is evaluated by measuring the code execution time of a set of integer micro-benchmarks. The results are compared to processor cores with different types of branch prediction.

3.7.1 Clock Frequency Study

The results for the Xilinx devices are based on Xilinx ISE 12.4 "place and route report". Likewise, the Quartus II tool-chain version 10.1 was used for Altera devices. Default synthesis and "place and route" settings are used for all FPGA families. In the Xilinx ISE tool-chain the global optimization parameter in the "map properties" is set to "speed". The frequency measurements of the

Altera devices are based on the Quartus II 85°C model. Table 4.1a lists the maximum clock frequencies and the required hardware resources of a Tinuso I implementation for various FPGA families. The design includes the cache controller implementation. Table 4.1b lists hardware resource usage and clock frequency with and without optional multiplier and barrel-shifter of Tinuso I and similar MicroBlaze configurations.

To allow for an unbiased comparison, the following features for MicroBlaze are disabled: exceptions, debugging infrastructure, divider, and pattern generator. The performance-optimized version of MicroBlaze with a 5-stage pipeline is used. Similar to Tinuso, MicroBlaze uses 4 kilo byte directly mapped caches implemented as block RAMs with a write-back strategy. Looking at the numbers of the minimal configuration in the upper part of the table, we observe that Tinuso operates at a significantly higher clock frequency than MicroBlaze while consuming 35% fewer resources.

The lower part of Table 4.1b shows hardware resource usage and clock frequency of a configuration with enabled multiplier and barrel-shifter. We notice that the implementation of multiplier and barrel-shifter lowers Tinuso I's clock frequency. Moreover, the costs to add these two functional units in terms of hardware resources are higher compared to MicroBlaze's implementation. These additional hardware costs are mainly caused by the fully pipelined 32-bit by 32-bit multiplier.

Intermediate results from the multiplier are passed over several pipeline stages, which is costly in terms of flip-flops. To avoid restrictions regarding forwarding and instruction scheduling we did not pipeline the barrel shifter. The large number of cascaded multiplexers that are used to implement the barrel shifter limit the clock frequency of the design. We conclude that the design principle of aggressive pipelining is very effective for the processor core. Extending the processor core with fast and complex functional units does not scale well for deep pipelines in FPGAs.

3.7.2 Branch Performance Study

The clock frequency results in Table 4.1 show that Tinuso takes optimal advantage of its deep pipeline. Superpipelined architectures where branches are resolved late in the pipeline have a high branch penalty. The performance of a processor architecture greatly depends on its ability to reduce the branch penalty. To evaluate Tinuso's branch performance we compare the code execution time of a set of micro-benchmarks to similar processor configurations. Commonly, there are minor differences in the way synthesizable processors im-

Table 3.1: Speed and resource overview of a Tinuso I implementation for various FPGA families

FPGA family	Grade	max. Freq.	Area
Xilinx Spartan 6	-3	220 MHz	1409 LUT-6
Xilinx Virtex 5	-3	335 MHz	1599 LUT-6
Xilinx Virtex 6	-3	376 MHz	1322 LUT-6
Altera Cyclone IV E	C6	206 MHz	1882 LUT-4
Altera Stratix III	C2	335 MHz	1332 LUT-6

Table 3.2: Overview of hardware resource usage and clock frequency of Tinuso I and MicroBlaze

minimal configuration	
Tinuso	MicroBlaze
376 MHz 1322 LUTs	194 MHz 2024 LUTs
4 BRAM 0 DSP48E	12 BRAM / 0 DSP48E
incl. multiplier & barrel shifter	
Tinuso	MicroBlaze
349MHz 2145 LUTs	194 MHz 2277 LUTs
4 BRAM 4 DSP48E	12 BRAM 4 DSP48E

plement arithmetic, logic, load, and store instructions. Major differences are found in the way processors minimize the negative effects of branch instructions. Hence, we decided to compare Tinuso I with Xilinx MicroBlaze 8.00b, MB8, and LatticeMico32, LM32. Tinuso uses predicated instructions to circumvent costly pipeline stalls due to branches. MicroBlaze utilizes dynamic branch prediction implemented with a branch target buffer. In the optimal case, it completely removes the average branch penalty [120]. LatticeMico32 supports static branch prediction only. Hence, always the same direction for the same branch is predicted.

The performance of large benchmarks suites, such as SPEC, highly depends on the performance of the memory hierarchy. To evaluate branch performance, we instead decided on micro-benchmarks that are independent of the memory hierarchy and show a high percentage of branch instructions. Hence, program and data are loaded into the caches in advance and a processor configuration without multiplier and barrel-shifter was chosen.

We measure the number of clock cycles each processor requires to execute

the benchmarks. To allow a consistent comparison among the three processor cores, both MB8 and LM32 are configured to match Tinuso I. LM32 has a barrel-shifter since its configuration tool-chain does not allow to disable it. MB8 and LM32 toolchains come with a port of GNU Compiler Collection, GCC. The highest compiler optimization level is used. At the time these results experiments were conducted, there is no operational C compiler for the Tinuso architecture available yet. Hence, we used the assembler output of the Xilinx GCC port as a starting point and adapted the code to match the Tinuso instruction set architecture. Loop unrolling optimization is not applied. This method effectively reduces the number of branches. Thus, it is not suited for the intended evaluation of Tinuso's approach to avoid branches by making use of predicated instructions.

The data type of all variables in all benchmarks is set to unsigned 32-bit integer. Even though Xilinx and Lattice tool-chains both use the same GCC compiler, in some cases the quality of the generated executable code varies significantly. For that reason the GCD application for the LM32 is written in assembler to ensure a consistent evaluation. We added performance counters to the processor configurations to record the number of clock cycles used to execute the benchmarks. The overhead to trigger the counter is subtracted from the results. We use the following micro-benchmarks:

GCD. The algorithm to determine the greatest common divisor, GCD, consists of a while loop, an if-then-else statement and two arithmetic operations where one operand is subtracted from the other or vice versa. We decided for an operand pair that leads to a total of 500 iterations. **RSA encryption**. The Rivest, Shamir and Adleman, RSA, encryption algorithm includes a multiplication and a modulo operation. Again, integer multiplier and divider are disabled on all of the three processors. Hence, the multiplication and the modulo operation are implemented with loops. A prime number pair of 37 and 3713 was chosen to yield a high number of iterations. **Fibonacci**. The Fibonacci algorithm is implemented without recursive function calls. The 40th Fibonacci number is computed. The structures of Fibonacci and GCD algorithms are comparable. However, the number of iterations in this benchmark is lower and there are slightly more arithmetic instructions within the loops. Hence, the impact of dynamic branch prediction is low. **Binary search**. The binary search algorithm locates the position of an item in a sorted array. Again, this array is loaded into the data cache before starting the benchmark to avoid a high number of data cache misses. An array size of 512 items is used. Hence, the number of iterations is relatively low, which limits the impact of dynamic branch prediction.

Figure 3.6a illustrates the relative performance of the three processors with normalized clock frequency. For the GCD benchmark, 50% of the instructions that MB8 and LM32 execute are branches. Hence, dynamic branch prediction provides a substantial boost in code execution performance. Tinuso leverages predicated instructions to implement if-then-else statements without using branch instructions. However, the loop iterations do not contain instructions enough that Tinuso can fill all branch delay slots. The LM32 requires most clock cycles to execute this benchmark since a taken branch requires four clock cycles [71]. Again, due to the high number of iterations of the RSA benchmark, dynamic branch prediction plays a vital role. However, the LM32 performs best on this benchmark. The fact that the LM32's configuration includes a barrel shifter explains this result.

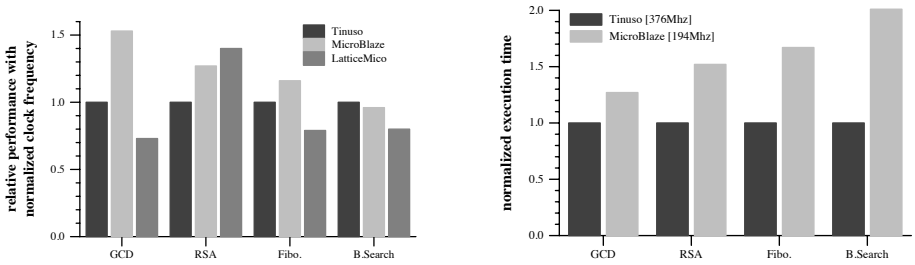
The number of iterations for the Fibonacci benchmark is lower than for GCD algorithm. Hence, the deviation of the results among the three processors is lower than for the GCD benchmark. Compared to the previous benchmarks, the main loop in the binary search algorithm contains more instructions. That allows Tinuso to fill the branch delay slots and to utilize predicated instructions effectively. Consequentially, Tinuso reaches the highest performance for this benchmark.

We observe that MicroBlaze performs best due to dynamic branch prediction. Some loop iterations do not contain instructions enough that Tinuso can fill all branch delay slots. Hence, Tinuso requires more clock cycles to execute these benchmarks than MicroBlaze. Nevertheless, Tinuso's predicated instructions allow for higher instruction level parallelism than LatticeMico32 where branches cause pipeline stalls.

To calculate the execution time of each benchmark, we scale the measured number of clock cycles, used to execute the benchmarks, with the processor's maximum clock frequency. We use the clock frequencies from Table 1 for Virtex 6 devices. LatticeMico32's product brief states a maximum clock frequency of 115 MHz when implemented on a LatticeECP3 FPGA [70]. Since it is difficult to compare designs on FPGA families from different vendors, we do not include LM32 in this study.

As seen from Figure 3.6b, Tinuso executes all benchmarks faster than MicroBlaze. Tinuso's high clock frequency allow for significantly better performance. Even on benchmarks where dynamic branch prediction has a big impact such as GCD, Tinuso performs best. The arithmetic average of the four benchmarks shows that Tinuso executes the benchmarks 38% faster than MicroBlaze. We show that our architectural design principles perform well. Tinuso's predicated instructions are an efficient way to circumvent costly pipeline stalls. This

approach can allow for a higher instruction level parallelism than conventional approaches where branches cause pipeline stalls.



(a) Performance with normalized freq.

(b) Execution time relative to Tinuso I

Figure 3.6: Performance results

3.8 Conclusions

In this chapter we described the architecture of Tinuso and evaluated the integer implementation of one instance of the Tinuso family architecture. Tinuso is a single-issue, in-order processor that delivers a high performance through a deep pipeline and uses predicated instructions to effectively circumvent costly pipeline stalls due to branches. Tinuso makes use of pipelined RAMs found in modern FPGAs to implement fast caches and register file. To attain highest clock frequency we pipelined the execution stage and designed a full forwarding mechanism that effectively minimizes the logic in the time critical path of the design.

Tinuso takes optimal advantage of its deep pipeline. We measured clock frequencies as high as 376 MHz on current FPGAs. The logic is well balanced between pipeline stages and the time critical path of the design includes only 4 successive 6-input lockup tables. Routing delays account for a substantial part of the critical path. Adding more pipeline stages is not beneficial as additional hardware resources and increasing complexity in the control logic lead to a diminishing return.

Tinuso is also effective in terms of hardware resources: A basic Tinuso implementation consumes 35% less area than an equivalent MicroBlaze configuration. We extended Tinuso with a multiplier and barrel-shifter and observed that

adding fast and complex functional units is very expensive in terms of resources for deep pipelines in FPGAs. The high clock frequency of a superpipelined processor comes at the cost of a high number of branch delay slots. We leverage predicated instructions to improve branch performance. We evaluate Tinuso's performance with a set of numerical and search-based micro-benchmarks and compared it to current state- of-the-art synthesizable processor cores. We achieve an average performance improvement of 38% over a similar MicroBlaze configuration.

We show that a superpipelined processor exploits the FPGAs architecture better than the classical 5-stage RISC pipeline. We demonstrate that our predicated instruction set architecture allows for higher instruction level parallelism than conventional approaches where branches cause pipeline stalls. We reach a higher performance while utilizing fewer resources than current commercial processor cores. Hence, the proposed design principles allow next-generation synthesizable processor cores to become more competitive.

Chapter 4

Tinuso Toolchain

This chapter describes a collection of tools to translate programs written in C programming language to executable binaries for the Tinuso architecture. This sequence of tools is called toolchain and consists of a C compiler, a collection of binary tools, and a C library. The architecture of Tinuso implements a deep pipeline to attain a high system clock frequency. All types of hazards are exposed to software to keep the hardware footprint low and predicated instructions circumvent costly pipeline stalls due to branches. Therefore, the Tinuso instruction set architecture requires tools and libraries to be adapted. For example, the toolchain must be able to resolve hazards, fill delay slots, leverage predicated execution, and emulate instructions that are not implemented in hardware such as floating point instructions.

Section 4.1 introduces the toolchain and describes main challenges that come with Tinuso architecture. Section 4.2 provides an introduction of the GNU GCC toolchain such as C compiler, Binutils and introduces the Newlib C library. Section 4.3 describes the GCC backend implementation for Tinuso. Section 4.4 evaluates the backend by measuring the code execution time for a set of small C benchmarks. Finally, Section 4.5 summarizes and concludes this chapter.

4.1 Introduction

Tinuso is a statically scheduled, pipelined processor architecture optimized for high throughput when implemented on FPGAs. A three operand, load-store instruction set with a fixed 32-bit instruction word length is used. Tinuso supports

a register file with 128 general purpose registers. To keep the implementation simple, Tinuso only issues a single instruction at the time and executes instructions in-order.

Register file size, cache sizes and cache organization are chosen to match the memory resources found in modern FPGAs. For example, Tinuso uses pipelined cache and register file accesses, which results in a deep pipeline. Therefore, branch instructions in Tinuso come with four branch delay slots, which make these instructions expensive. Predicated execution is a technique, which enables the elimination of branch instructions. A predicated instruction uses a predication register as an additional operand. Depending on the value of the predication register the instruction is executed or annulled, i.e., has no effect. Tinuso comes with eight predication registers that are a subset of the 128 general purpose integer registers. The Tinuso architecture leverages predicated execution for all instructions except of movhi.

Simple if-statements can often be replaced with a sequence of predicated instructions. This method is called *if-conversion*. The better a compiler is able to apply if-conversion, the more branch instructions are removed, which lowers the total amount of branch delay slots in the program code. Moreover, Tinuso's predicated instruction set enables selectively annulling of instructions in delay slots. Hence, it is possible to fill branch delay slots with instructions from before the branch, from the branch target and from fall through.

To keep the hardware footprint low, the pipeline is fully exposed to software where a compiler needs to resolve all types of hazards. The current Tinuso implementation supports forwarding from all pipeline stages and most instructions execute without any pipeline hazards. However, the design is optimized for a high system clock speed, which prevents some instructions to complete the computation in the second execution stage. This causes additional delay slots and scheduling restrictions a compiler has to consider. Table 4.1 provides a brief overview of delay slots of common instructions. Moreover, there is a forwarding limitation for a certain sequence of predicated instruction as described in Section 3.6. The current cache controller implementation also restricts instruction scheduling as it does not permit load and store instructions to be placed in branch delay slots.

It is not clear if current production compilers can successfully use predicated execution and schedule instructions so as to mitigate the delay slots. In this chapter we describe the development a GCC, *GNU Compiler Collection*, backend for Tinuso [108]. GCC's instruction scheduler and its ability to exploit predicated execution are evaluated for Tinuso's exposed 8 stage pipeline. We implement a GCC 4.8.1 compiler toolchain for the Tinuso architecture and de-

Table 4.1: Delay slots for a Tinuso 8 stage pipeline

Instruction	delay slots
branch / jump	4
load word	4
store word	3
compare-and-set	2,(1)
barrel shift	2

scribe the architecture specific scheduling optimizations and how we leverage predicated execution. Finally, we run a set of small C benchmarks and compare the code execution time to a similar Xilinx MicroBlaze configuration.

A GCC compiler toolchain is used because it is an open source compiler that supports a large number of target platforms and has built-in support for predicated execution [108]. GCC yields better performance than LLVM on single issue in-order pipeline [63]. Moreover, GCC is widely used for embedded systems. For example, companies such as Altera, Lattice, Microchip, TI, and Xilinx provide GCC based toolchains for their embedded processors. GCC has a modular structure and has an infrastructure for describing new target architectures.

4.2 GNU Compiler Collection Overview

A compiler transforms programs written in a programming language into another language. The target language is most often machine code or assembly code for a target architecture. GCC is a collection of compilations tools that was initially written by Richard Stallman in 1987 as part of the GNU project. Meanwhile, GCC supports a total of 7 programming languages and more than 30 target architectures [1, 2]. GCC is published under the GNU Public License, *GPL*, which allows to freely use, modify and to re-distribute the source code. GCC is popularly used in embedded computing where most producers of embedded processors provide GCC based toolchains.

Figure 4.1 shows an overview of the Tinuso toolchain. The compiler reads in program code in C programming language and outputs Tinuso assembler code. The GNU Binutils are then used to create executable binaries for the Tinuso architecture. The GNU assembler, *gas*, translates incoming assembler code into relocatable binary object code. This object code consists of a sequence

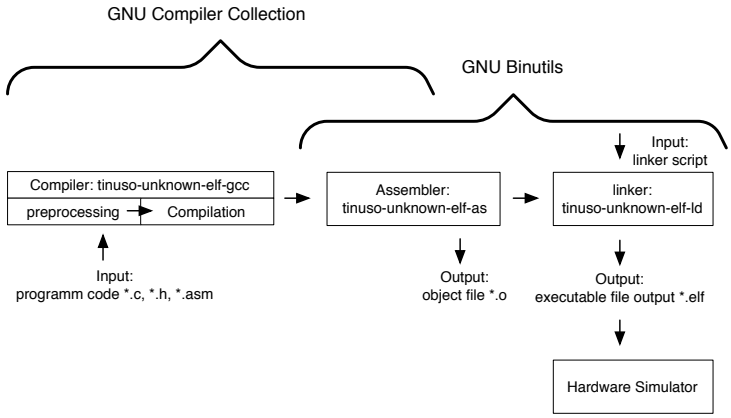


Figure 4.1: Tinuso toolchain overview

of machine specific instructions and symbolic links to data or objects and object files of other program code. The C library is compiled with GCC toolchain into a set of object files that are then copied into a static library file. Finally, the linker collects object files and libraries, resolves symbolic links and generates executable binaries for the Tinuso architecture.

Most modern production compilers can be split up in three main parts. First, the frontend of a compiler scans the program code, parses its syntax and semantics and translates it into an intermediate representation form. Second, the middleend performs a number of optimizations passes by rearranging the code. Third, the backend chooses target instructions to match each instruction of this intermediate representation. The backend is responsible for low-level and machine specific optimizations, instruction scheduling, register allocation and output of assembly code. Therefore implementation details of the target architecture need to be described in the back-end. This structured architecture allows for adding new programming languages by adapting the frontend only. The internal representation allows for adding optimization passes that are independent of programming language and target platform. Additional target architectures can be supported by extending the backend only.

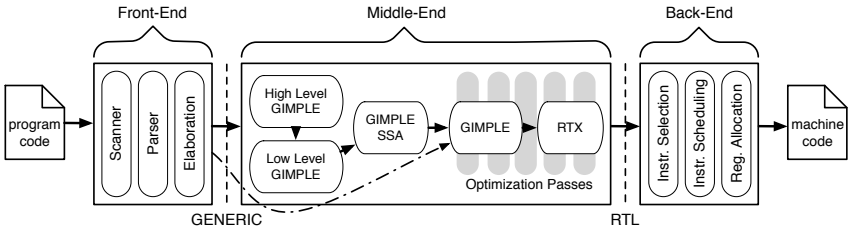


Figure 4.2: Struture of GCC with intermediate representations

4.2.1 GCC Intermediate Representation

Figure 4.2 shows the GCC infrastructure and its intermediate representations. GCC uses three main intermediate representations during compilation: GENERIC, GIMPLE and RTL [108]. GENERIC is a language-independent representation generated in the frontend. It is able to represent programs written in all the languages supported by GCC. GENERIC is an abstract-syntax tree that serves as an interface between the frontend and middleend. However, a frontend may also directly generate GIMPLE as the C and C++ frontends do.

GIMPLE is a language independent, tree based representation. GIMPLE is a simplified subset of GENERIC that is more restrictive. For example, GIMPLE does not allow expressions with more than three operands [82]. Target and language independent optimizations such as inlining, constant propagation, tail call elimination, and redundancy elimination are done in GIMPLE. Most of the work of GCC is done on a low-level intermediate representation called register transfer language, *RTL*. RTL is inspired by Lisp lists to describe the instructions to be outputted in an algebraic form. GIMPLE and RTL representations are used to optimize the program. In total, GCC 4.8.1 comes with more than 250 optimization passes.

In addition to the main intermediate representation GCC uses three subtypes of GIMPLE. High and low level GIMPLE are produced when the compilers converts the program representation from GENERIC to GIMPLE. This conversion is called Gimplification. To allow for efficient program analysis, GCC rewrites the low level GIMPLE in the static single assignment form, *SSA*. SSA is a property of an intermediate representation that strictly requires a unique definition for each variable. The advantage of intermediate representations in SSA form is that certain compiler optimizations are more efficient and execute faster.

4.2.2 GCC Frontend

Figure 4.2 shows the GCC infrastructure including the high level tasks to be performed in front-, middle-, and backend.

Before a compiler can optimize expressions and translate them into machine code, it must understand both the programs syntax and its semantics. Hence, the frontend first determines if the program code is valid and then, in a second step, generates a tree based intermediate representation of the program code. Finally, the tree based intermediate representation is then passed to the middleend of the compiler.

The input of the compiler frontend is program code written in a specific programming language. GCC supports programming languages such as C, C++, Objective C, Fortran, Java, Ada, and Go [52]. The first step the frontend performs is to scan the program code. The scanner reads a stream of characters and breaks it down into small atomic units called tokens. Each token is a single unit of the source programming language, for instance a keyword, identifier or symbol name. During the lexical analysis a set of rules are applied to determine whether or not the tokens are legal in the source language. Some programming languages, such as C, execute a preprocessing phase where macros are substituted and conditional compilation is resolved. This task is typically performed before the syntax and semantical analysis.

The parser transforms the sequence of tokens from the lexical analyzer into in a parse tree according to the syntax rules of the input language. This parse tree allows for identifying the structure of a program and to check whether or not the program code fulfills the grammar of the input language. Type checking is also performed during this stage of the compilation process. After semantic analysis has completed successfully, these language-specific parse trees are lowered to the language-independent GENERIC representation. This is done by replacing high-level, language-specific trees with lower-level equivalents.

4.2.3 GCC Middleend

The middleend of a compiler typically performs optimizations on an intermediate representation of a program that is independent both of input language and target architecture. GCC first runs the Gimplification pass to convert the intermediate representation of a program from GENERIC into GIMPLE. Then, GCC performs optimizations on the GIMPLE program representation in SSA form. The optimized GIMPLE is converted into RTL and some more optimizations passes are executed. Finally, optimized RTL is handed over to the

backend.

The input of the GCC middleend is program code in an intermediate representation. Depending on the frontend GENERIC or GIMPLE is used. In a first step GCC runs the Gimplification pass to convert GENERIC into GIMPLE. In a next step, GCC rewrites the GIMPLE representation into the static single assignment form, *SSA*. The SSA form requires that variables are assigned not more than a single time. As in actual programs variables often are assigned multiple times GCC creates new versions of these variables. GCC optimization passes require the SSA form as certain compiler optimizations execute faster, optimization are more effective, and the SSA form eases the development and maintenance of optimization passes [3, 51]. There are optimization passes that add new symbols or change the program variables and may invalidate the SSA property. Hence, the pass that rewrites GIMPLE in the SSA form may be called multiple times throughout the optimization process.

The following briefly describes some optimizations passes that GCC performs on the GIMPLE SSA representation. The complete list of optimization passes can be found in the GCC internals documentation [108].

- The dead code elimination pass scans the program code for statements that produce unused results. This pass is not able to check whether data stored in memory are used again later or not, all memory locations are therefore always considered as used.

- The profiling pass inserts functions into the program code that collect run time profiling data. This data may be fed back to the compiler on a subsequent run to perform optimizations that consider expected execution frequencies and access patterns. For example, profiling data may be used for high level memory optimization to increase the memory hierarchy performance. The analysis of access patterns allows for reorganizing matrices to improve cache locality. [58].

- GCC performs a high number of loop optimization such as vectorization, loop-nest optimization, and loop unrolling. To exploit data parallelism, the vectorization pass transforms loops to operate on vector types instead of scalar types. Grouping data elements from consecutive iterations into a vector that allows for computing loop iterations in parallel. Loop-nest optimizations aim for improving data locality and removing dependencies that might prevent other optimizations. Loops with few iterations may be unrolled.

- The array prefetching pass issues fetch instructions inside loops to prefetch data for future loop iterations.

Once all optimizations on the GIMPLE representation have completed, the program code is converted into low-level RTL intermediate representation. This process is called RTL expansion. The conversion is done in two steps. First GCC converts the representation back to GENERIC and eliminates as many temporary variables as possible. Second, the tree representation is converted into standard named patterns that describe low level operations. These patterns then are used to generate RTL. The RTL representation is a chain of objects called *insns*. Each target architecture must have a number of standard patterns defined to allow RTL generation. On some target machines such as Tinuso, some of these standard patterns cannot be expressed with single insn. In such cases, a `define_expand` pattern can be used to describe a sequence of insns that form a RTL pattern.

According to Figure 4.2, there are RTL optimizations done both in middleend and backend. The program representation in RTL at this point is still independent on the target architecture. Hence, RTL optimizations that are independent on the target architecture are therefore considered to be part of the middleend. For example there are optimization passes to remove redundant computations that are not visible at GIMPLE level. The partially optimized RTL representation is then handed over to the backend for architecture dependent optimizations and machine code generation.

4.2.4 GCC Backend

The GCC backend receives the RTL representation of the program code and runs a number optimization passes. The backend is responsible for machine specific optimizations, instruction scheduling, register allocation, and to output assembly code. The last step is performed by matching the insn list against RTL templates to produce assembler code. GCC requires a detailed description of the target architecture for these tasks. This information is located in the machine description which includes a detailed description all instructions of the target machine in algebraic form. Moreover, the machine description contains target specific macro definitions to describe the target architecture. GCC code is target independent, but may depend on machine parameters such as addressing conventions. The strict separation between target independent program code and machine description ensures GCCs portability. If GCC shall support another target machine, it is sufficient to add a machine description of

this architecture.

The following briefly described the task performed in the backend:

- GCC runs an if-conversion pass that aims to replace conditional branches with predicated instructions. In a first step, this pass modifies conditional branches to depend on a Boolean value of a comparison instruction. Then, predicated instructions are used to replace the conditional branch when supported by the target architecture.
- The instruction combination pass attempts to combine the simple RTL patterns of the initially generated representation into more complex patterns and match the result against the machine description.
- There are target architectures where the output of an instruction will not be available before the next subsequent instructions may need it. For example Tinuso has such delay slots for memory loads, compare-and-set instruction, and barrel-shift and multiply operations. The instruction scheduling pass attempts to re-order instructions to avoid data hazards.
- Until now, the program representation was allowed to use any number of registers. The program representation may therefore include registers that physically exist on the target machine, called hard register, and registers that do not exist in hardware, called pseudo registers. The register allocation pass replaces these pseudo register by replacing them with hard register, pushing them on the stack or replacing them with equivalent expressions such as constants.
- Architectures with deep pipelines typically have multiple branch delay slots. Instructions in these delay slots are executed regardless whether the branch is taken or not. The delayed branch scheduling pass selects independent instructions that can go into these delay slots.
- The final pass outputs the assembler code. The assembly letters to output are defined in the machine description. The assembly output is then typically handed over to the GNU Binutils to produce executable machine code.

4.3 Tinuso GCC

The architecture of Tinuso comes with a high number of delay slots and scheduling restrictions and provides predicated instructions to reduce the branch penalty.

Hence, the performance highly depends on GCC's ability to schedule instructions for an exposed 8 stage pipeline and to leverage predicated execution. We have developed an optimizing GCC backend for Tinuso. This backend leverages the GCC infrastructure and can perform all common optimizations such as aggressive register allocation and proper instruction selection including conversion of expensive operations such as multiplications to less expensive operations such as additions and shifts.

Although GCC comes with a total of 30 officially supported target architectures, the architecture of Tinuso is in many ways different from other target machines, which results in large number of machine specific optimizations. The following briefly describes properties of the Tinuso architecture and the implications for the GCC backend.

- Tinuso comes with a very small instruction set. For example, addition is the only arithmetic or logical operation that includes an immediate field. Hence, all other operations that include constant operands need to be composed of instructions to load constants into a register and instructions to perform the computation. 32 bit constants are loaded with a `movhi` and a successive `addi` instruction. Tinuso implements an asymmetric immediate split (22 bit/10 bit) to increase the opcode space for operands. GCC require RTL standard patterns for 32bit, 16bit and 8bit data types in the machine description. Hence, many 16-bit operations cannot be mapped to Tinuso instructions directly, instead the backend has to generate instruction sequences to emulate these operations. Except of RISC-V, all current instruction set architectures implement a symmetric immediate split [114].

- Except of `movhi` all instruction of the Tinuso ISA are predicated. Hence the entire machine description must include definitions for conditional execution. Most target machines that support predicated execution work with predication flags while Tinuso comes with 8 predication registers. Texas Instruments c6x DSP architecture implements predicated instruction in a similar way. C6x comes with 2 sub register files of which each supports 4 predication registers [56]. However, the GCC backend for c6x makes only limited use of GCC's infrastructure for predicated execution. Instead, an architecture specific scheduling algorithm that enables predicated execution, fills the delay slots, and resolves hazards in the program code is implemented. This results in a very complex compiler implementation that requires a huge development effort and makes maintenance costly.

- Tinuso only supports memory operations in word length. The backend therefore needs to insert shift and mask instructions if shorter data types are accessed in memory. This is similar to early Alpha implementations without BWX extension [31]. All other architectures come with memory operations for data types shorter than word length.

- While most other target machines have hardware mechanisms to resolve hazards the Tinuso pipeline is fully exposed to software. Hence, the compiler has to insert nop instructions to resolve all types of hazards. GCC infrastructure comes with support to describe delay slots. RTL optimization passes such as instruction scheduling and delayed branch scheduling try to fill these delay slots. However, there are situations where optimization passes work against each other and a machine specific pass need to be implemented to circumvent the problem.

The GCC infrastructure provides a number of optimization passes common to all architectures. Two are particularly relevant for Tinuso's delay slots. First, the instruction scheduling pass arranges instructions according to data dependencies. This pass attempts to re-order instructions globally to avoid data hazards, based on architecture specific information about delay slots. However, control-flow instructions are not handled by the instruction scheduling pass. Control-flow instructions and their delay slots are instead handled by the delayed branch scheduling pass, which runs after the instruction scheduling pass.

Tinuso does not require an implementation of instructions such as integer multiplication, division or floating point operations. Therefore software implementations of these operations have been implemented using libgcc, a GCC library for support routines. We manually optimized the assembly routines for the most commonly used integer operations. To help the scheduling passes we explicitly make details on register usage on these hand-coded routines available to the compiler, allowing it to more freely schedule instructions and use more register across function calls. Moreover, we allow GCC to inline the integer multiplication routine. This often pays off because costly function calls are omitted and Tinuso's predicated ISA enables an efficient multiplication implementation. GCC comes with built-in support for predicated execution and provides an if-conversion pass. However, the if-conversion pass is limited in scope and only simple control flow patterns are handled.

Listing 4.1: RTL Addition instruction pattern

```
1  ;;name
2  (define_insn "addsi3"
3  ;;RTL template
4    [(set (match_operand:SI 0 "predicate" "constraint")
5          (plus:SI (match_operand:SI 1 "predicate" "
               constraints")
6                   (match_operand:SI 2 "predicate" "
                       constraints"))))]
7  ;;condition
8    ""
9  ;;output template
10   "@
11    add \%0,\%1,\%2"
12    addi \%0,\%1,\%2"
13 ;;attributes
14   [(set_attr "type" "arith_3op")
15    (set_attr "length" "4")
16    (set_attr "predicable" "yes")])
```

4.3.1 Tinuso Machine Description

This section describes the Tinuso machine description for the GCC backend. The machine description consists of a RTL based specification file of the target instruction patterns and a C header file of macro definitions. Additional target specific functionality is implemented in a C file. Typically, GCC backends also include Makefile fragments for target-specific options.

- Tinuso.md is the machine description file that describes each instruction of the target architecture in algebraic form. It contains a detailed descriptions of all instructions of the target machine and expand patterns that describe sequence of insns to form standard RTL patterns. Listing 4.1 shows the pattern of an addition instruction in Tinuso. `Addsi3` is the name of this pattern that describes a three operand addition. Machine modes are used to describe the size of data object and the representation used for it. For example SImode stands for "single integer" and typically represents a four-byte integer [108]. The RTL template defines which insns matches a pattern and how to locate the operands. In this example the RTL template defines that two operands in SI mode are added and

the result is stored in register. The RTL template includes predicates to determine whether an insn operand matches the RTL pattern. A pattern may include machine independent predicates that require the operand to be present in a register or machine specific predicates that allows the operand to be a constant value within a certain range. Constraints then used for fine-tune matching to select a pattern within the set of operands specified by the predicate. For example, Listing 4.1 constraints are used to determine whether an addition instruction or an addition immediate instruction is emitted. A pattern may also include a condition in the form of a C expression. This is often used to enable and disable target specific options such as hardware floating point operations [85]. The output template is used describe the assembler code of a matching insn. In the example of Listing 4.1 a addition is emitted. The %x in this string specifies the operand number. Finally, patterns may include attributes. For example Tinuso uses attributes such as instruction length, instruction type, and predicate which defines whether a certain emitted instruction may be predicated or not. This information is used for instruction scheduling.

- The predicates.md file includes machine specific predicates that are used to determine whether a operand matches a RTL patterns.
- The constraints.md file contains machine specific constraints that are used to match RTL operands to insn patterns. Tinuso only includes target specific integer constraints for instructions with immediates and offsets.
- Tinuso.h is a header file that contains machine specific Macro definitions such as register definitions, memory addressing mode, and ABI information.
- GCC provides target hooks for machine specific adjustments. These functions may be stored in tinuso.c file as well as advanced functions used in machine description patterns. The Tinuso backend implements functions to set up the stack frame, to compose conditional branches, to leverage predicated execution and for memory accesses of short data types.
- The tinuso-protos.h file contains prototypes for functions in the tinuso.c file.
- The tinuso.s file contains optimized assembly code for standard RTL patterns. Tinuso uses this for operations such as multiplication, division, and modulo.

Listing 4.2: Simple if statement in C programming language

```
1 if (A == 0)
2    {B++;}
3 else
4    {B--;}
```

- The t-tinuso file includes Makefile fragments for target-specific options. For example, in the Tinuso processor architecture operations such as integer multiplier and floating point unit are optional. Various hardware configurations can be expressed in the t-tinuso file. Based on this information the compiler knows which processor configurations implement optional operation in hardware. Hence, it emulates the missing optional operations in software. When a program is compiled, command-line options are set to produce executable binaries for a given processor configuration.

4.3.2 Memory Access

In 1989 MIPS patented unaligned memory accesses for RISC pipelines [50]. Hence, initial implementations of the Alpha architecture did not include memory operations of 8 and 16 bit data types. Instead, these data types are accessed using mask and shift operations. Tinuso implements this functionality with an expand pattern in the machine description for 8 and 16 bit moves. If there is a memory operand, the expansion patterns forces the other operand into a register and calls a C function, which emits a sequence of insn patterns to access the short data types in memory. This sequence is emitted during the RTL expansion, hence, successive RTL optimization passes and delay slot scheduling passes can optimize these memory access sequences respectively insert nops if required.

4.3.3 Predicated Instructions

Tinuso's predication scheme was designed with the perspective that a future C compiler can make the best possible use of predicated execution. Consequently most instructions are predicated, multiple predication registers are available and the predication condition can be inverted. If-conversion is a method where simple if-statements are implemented with a sequence of predicated instructions. Control flow dependencies are converted into data dependencies to avoid costly

Listing 4.3: Assembly code on Tinuso without predicated execution

```
1  bz rA label1
2  nop
3  nop
4  nop
5  nop
6  addi rB, rB, -1
7  bz r0 label2
8  nop
9  nop
10 nop
11 nop
12 label1:
13 addi rB, rB, 1
14 label2:
```

Listing 4.4: Assembly code on Tinuso with predicated execution

```
1  [!rA] addi rB, r0, 1
2  [rA]  addi rB, r0, -1
```

branch instructions. The following example, Listing 4.2 shows a simple C if-else construct.

If the compiler does not leverage predicated execution, two branch instructions are required to implement simple if-else statements. The assembly output for Tinuso without predicated execution is shown in Listing 4.4. When if-conversion is applied, branches are removed which results in a much more efficient implementation. Listing 4.4 shows the assembly for Tinuso with predicated execution.

GCC comes with built-in support for predicated execution and provides an if-conversion pass. GCC infrastructure supports to describe predicated executions. Tinuso's RTL implementation to enable predicated execution is shown in Listing 4.5.

This particular construct enables predicated execution if a predication operand in a predication register `pred_operand` is not equal, `ne` , to zero `const_int 0`. Insn patterns that support predicated execution need to be marked with the

Listing 4.5: RTL description of predicated execution

```
1 (define_cond_exec
2         [(ne (match_operand:SI 0"pred_operand" "r")
3         const_int 0))]
4         ""
5         [!%0]")
```

`predicable` attribute set to `true`. The if-conversion pass for Tinuso ensures that the result of the if-condition, `A == 0` from the example above, is copied into one of the eight predication register. In many cases, the GCC infrastructure is then able convert simple if statements to straight line code with predicated instructions. However, this works only if the control flow branches to the subsequent basic block. This is a major limitation as GCC does not perform basic block scheduling and optimize predicates during if-conversion, which have previously been shown to be very powerful and make it possible to convert many forward and backward branches [80, 92].

GCC also supports annulled branches where the predicated instructions are used to annul instructions in branch delay slots depending on whether the branch is taken or not. However, GCC requires annulling of all instructions in branch delay slots, which is a major limitation for the Tinuso architecture because two reasons. First, it can only rarely be applied as it requires that none of the delay slots is filled. Second, Tinuso does not allow instructions with delay slots to go into the delay slots of other instructions. Therefore there are only a limited number of predicated instructions that can be put into delay slots. The compiler backend therefore currently does not make use of annulled branches. However, Tinuso would greatly benefit form a annulled branch optimization pass that is able to fill branch instructions with a mixture of instructions from before the branch, from the branch target, and from fall through.

The current compiler backend follows a convention that `r0` is always set to 0. Thus, for instructions that do not use predication, the assembler inserts r0 as predication operand and sets the negate bit. Hence, these instructions are always executed. Future work on the compiler may remove this convention to give the compiler more freedom with predicated instructions. In the header file, there is an C expression `max_conditional_execute` that defines the maximum number of predicated instructions to execute instead of a branch. For Tinuso this is set to five, which corresponds to cost of a single branch instruction.

Listing 4.6: RTL description of delay slots

```
1 (define_insn_reservation
2               "compare-and-set" default_latency: 3)
```

Listing 4.7: RTL description of branch dealy slots

```
1 (define_delay branch / jump
2         [delay-1 (nil) (nil)]
3         [delay-2 (nil) (nil)]
4         [delay-3 (nil) (nil)]
5         [delay-4 (nil) (nil)]
```

4.3.4 Delay Slot Scheduling

The performance of Tinuso highly depends on the instruction scheduler ability to mitigate the delay slots. The GCC infrastructure provides two optimization passes that are particularly relevant for Tinuso. First, the instruction scheduling pass arranges instructions according to data dependencies and inserts nops to avoid hazards. Second, the delayed branch scheduling pass attempts to fill delay slots of control flow instructions.

GCC provides infrastructure to describe the pipeline characteristics of an instructions. Each instruction with a delay slot, except of control flow instructions, require such an `insn_reservation` that defines the number of clock cycles until the result of the given instruction is available. Listing 4.6 describes Tinuso's delay slots of compare-and set instructions.

The instruction scheduling pass arranges instructions according to data dependencies and attempts fill as many delay slots a possible and inserts nops if necessary. Delay slots of control-flow instructions are handled by the delayed branch scheduling, which runs after the instruction scheduling pass. A `define_delay` with structure as shown in Listing 4.7 is used to describe the delay slots of control flow instructions in Tinuso.

This construct defines 4 delay slots, that the compiler attempts to fill with useful independent instructions. It allows for specifying which instruction types may go into which delay slot. Typically instructions form before the control flow instruction are used to fill the delay slots. However, it is possible that the delayed branch scheduling pass picks instructions that were placed by the

instruction scheduling pass and thus reintroducing hazards. This is obviously a major limitation to the built-in GCC passes that requires a machine specific pass need to be implemented to circumvent the problem.

Architectures such as c6x circumvent this problem by not using GCC's delayed branch scheduling pass and implementing architecture specific functions to fill branch delay slots. As implementing a new scheduling pass is a daunting task and problems with conflicting passes are relatively rare, we use GCC's built-in passes but add a small simple pass in the Binutils that just detect and resolve problems introduced by the delayed branch scheduling pass by inserting nop instructions.

4.3.5 Tinuso GNU Binutils

The GNU Binutils are a set of low level programming tools that are typically used in conjunction with GCC compilers. These tools allow for creating and managing binary programs, object files, libraries, profile data, and assembly source code. In particular, Binutils includes the following main components: support libraries such as *BDF* and *opcodes*, the assembler *gas* to transform assembly into object code, and the linker *ld* to group object into executable binaries. There are tools to auto-generate new targets for GNU Binutils such as CGEN and ArchC [12, 67]. Unfortunately, Tinuso's syntax for predicated instructions is supported neither by CGEN nor ArchC. It was therefore necessary to port the Binutils manually to support the Tinuso architecture.

- Opcodes is a library of instructions of a given processor architecture. It describes the assembly syntax and instruction set encoding, which is used by both assembler and disassembler.

- The Binary File Decriptor library, *BFD*, contains a set of common routines to manipulate object files. The binary format of object files consists of four parts. First, a file header that includes general file information and a set of pointers to other parts of the object file. The second part contains a number of sections of raw code data. The third part consists of relocation tables. Finally, symbol tables are placed in the fourth part.

- The Tinuso assembler has two main functions: it translates assembly code to machine code and it checks the assembly and inserts no-operation instructions to avoid hazards in the pipeline. In principle, GCC is fully responsible for instruction scheduling, but there are situations where it is simpler to detect and resolve hazards in the Binutils. This is implemented

in Binutils with a buffer that includes the latest four emitted instructions. However, we aim for a solution where all hazards are resolved in RTL the compiler backend. GCC performs a large number passes on the RTL representation that rearrange instructions to find a global optimum. Hence, it is difficult for GCC to find a global optimum when afterwards the Binutils insert instructions. For example, GCC is able to align instructions to improve cache performance, when the Binutils then later add instructions it might negatively affect the cache performance.

- As a final step in the compilation process, the linker collects object files and creates an executable binary. The linker is responsible for resolving memory addresses, called relocations, and symbol references. It parses object files and libraries and replaces symbolic references or names of libraries with actual memory addresses. A linker script defines memory regions where individual sections of the program are placed. For example, an embedded system may include RAM and ROM, the linker script ensures that read-only data is placed in the address space of the ROM.

4.3.6 Tinuso C Library

Newlib is an open source C library designed for embedded systems that is maintained by Red Hat. Newlib can easily be ported to new processor architectures and runs with or without operating systems. In Tinuso there is currently no operating system available, hence, the C library cannot make use of system calls. Instead, Tinuso comes with a board support package for Newlib that implements basic low level functionality such as basic I/O, timer and file handling [14].

4.4 Toolchain Evaluation

The GCC 4.8.1 based backend for Tinuso is evaluated by compiling and running a set of small C benchmarks. We compare the execution time on Tinuso with a similar MicroBlaze configuration. However, for the MicroBlaze configuration the Xilinx's most current tool set, which is based on GCC 4.6.4, is used. The experiments are conducted with a Tinuso configuration where integer operations such as multiplication, division, and all floating point operations are emulated in software. This leads to a large number of function calls and costly control flow instructions. The analysis of the instruction mix then provides an insight into GCC's use of predicated execution and the scheduler's ability fill delay slots.

The experiments include two Tinuso configurations: First, a single core setup with the memory controller directly connected to the cache controller is composed. Second, a network-on-chip, *NoC*, configuration with a packet switched interconnect between processor core and memory controller as described in Chapter 5 and Chapter 6 is used. There are different memory latencies for the two systems as the latter system includes a network interface that composes and decodes packet messages on cache misses. A cache miss in the setup with integrated memory controller is resolved within 17 clock cycles while it takes up to 35 clock cycles to resolve a cache miss in the NoC setup.

A set of applications from the WCET benchmark suite [49] is used. These benchmarks are applications that operate on small data sets and do not require OS support. COMP is a data compression program adopted from the SPEC95. DES is a complex embedded program that includes a lot of bit manipulations. FIR is a signal processing algorithm, which computes the finite impulse response on a small data sample. Prime calculates whether two large numbers are prime numbers. Qsort implements a non-recursive version of the quick sort algorithm. Minver computes the inverse of a matrix of floating point numbers. FFT is a fast Fourier transformation on 1024 data points. LU decomposition factors a floating point array with 50 elements. FFT, LU, Minver, and Qsort benchmarks include floating point operations.

To evaluate the compiler toolchain we run the same benchmarks on a similar MicroBlaze configuration. The results refer to MicroBlaze 8.50.b, which is part of the most recent Xilinx ISE Design Suite 14.6. The compiler toolchain for MicroBlaze is based on GCC 4.6.4. To allow an unbiased comparison, the MicroBlaze is configured similarly to Tinuso with a barrel shifter while hardware multiplication, division, and floating point units are disabled. Tinuso and MicroBlaze configurations both use a 16 kilo byte directly mapped instruction cache and a 4 kb data cache with a write-back strategy. Smaller instruction cache sizes led to collision misses for the benchmarks that use GCC's double precision floating point library.

In addition, all benchmarks are run on two additional MicroBlaze configurations. One with hardware integer multiplication enabled and another configuration with additional floating point unit enabled. All MicroBlaze configurations are synthesized to hardware and the applications are run directly on hardware. Applications are recompiled to make use of all hardware available in each configuration. While Tinuso is fully synthesizable to both Altera and Xilinx FPGAs, we simulate the Tinuso's VHDL sources to run and profile the benchmarks for each Tinuso system.

The MicroBlaze setups make use of DDR2 main memory. For Tinuso, we

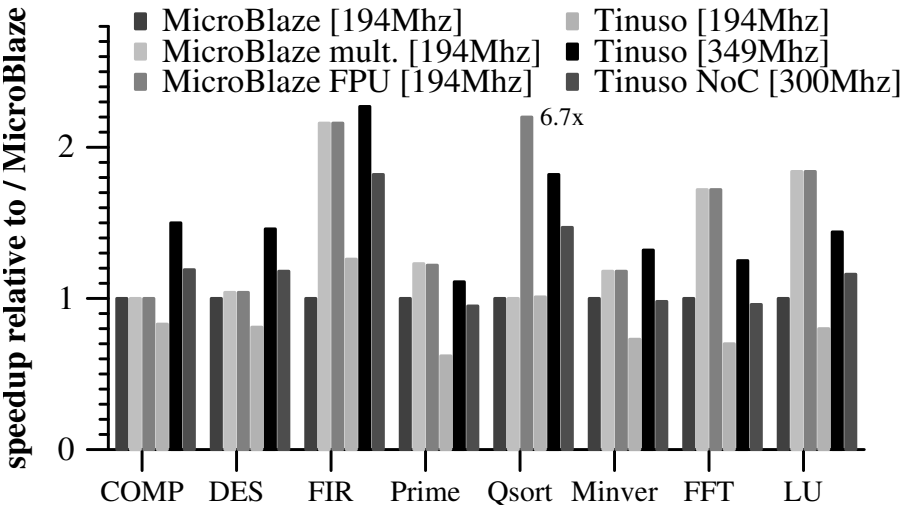


Figure 4.3: Speedups relative to a MicroBlaze configuration without hardware support for multiplications or floating point operations

simulate the same memory hierarchy. There is, however, a difference in memory performance between Tinuso and MicroBlaze. On MicroBlaze cache misses incur a penalty of about 10 clock cycles only.

For all benchmarks, the optimization level -O3 is used. Figure 4.3 shows the code execution time relative to the minimal MicroBlaze configuration. Number of clock cycles, used to execute the benchmarks is measured and scaled the results with the processor's maximum clock frequency [105]. We observe that MicroBlaze's integer multiplication unit has a positive impact for FIR, Minver, LU, and FFT benchmarks. The hardware floating point unit only pays-off for the Qsort benchmark, which executes many single precision floating point operations. Both FFT and Minver operate on double precision floating point numbers, which are not supported by MicroBlaze's single precision only FPU. Instead, the double precision operations are carried out in software.

Tinuso shows the best performance for FIR. FIR includes a high number of integer multiplication and divisions. These routines are very efficient on Tinuso because they exploit predicated execution to remove many branch instructions and most delay slots are filled. On top of that, our backend implementation is aware of the register usage on these routines, which results in more efficient

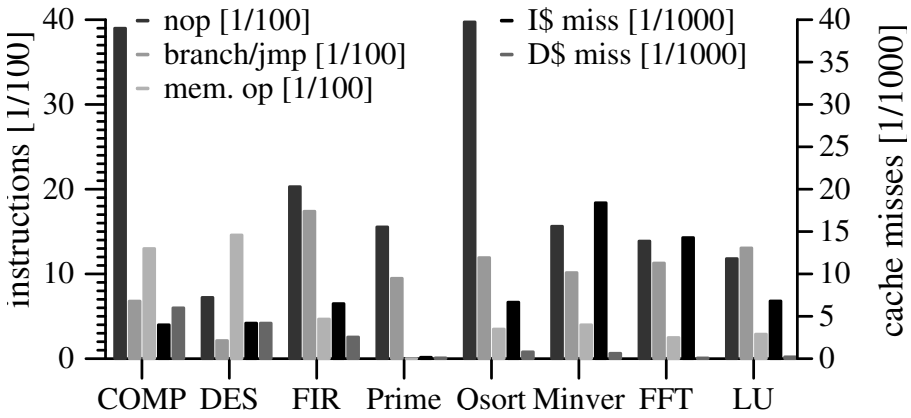


Figure 4.4: Benchmark analysis

register allocation across function calls.

The Tinuso NoC setup executes the benchmarks on average about 22% faster than the similar MicroBlaze configuration. We observe the lowest performance for Prime and Minver. Prime implements a small loop-nest that includes branches. GCC is not able to apply if-conversion on the control flow patterns of Prime. The performance for Minver is low because of the high number of cache misses. The Tinuso setup without NoC reaches an average speedup of 52% over a similar MicroBlaze configuration.

Figure 4.4 shows the frequency of occurrence of nops, control-flow instructions and memory operations. We observe a very high percentage of nops in the COMP and Qsort benchmarks. COMP frequently executes a function that loads and analyzes the input data in 8-bit portions. This leads to code with frequent compare and branch related delay slots and there are not enough independent instructions to fill the delay slots. Moreover, the control flow patterns of this function are not handled by GCC's if-conversion pass.

Qsort executes a high number of calls to floating-point compare functions where we observe the same behavior as described above. The other benchmarks include between 7% and 20% nops, which reflects the number of branch instructions. The FIR benchmark performs well even though it has the highest number of branch instructions. For FIR we record that 12.7% of all executed instructions make use of predication. FIR uses more predicated instructions than the

other benchmarks, which leads to a good performance.

The complete instruction mix of all benchmarks is shown in Table 4.2. On average over all benchmarks only about 6% of all executed instructions make use of predication. Given the high number of nops in the code and the low percentage of predicated instructions, there is room for improvements with respect to predicated execution.

Our main purpose of the experiments was to analyze instruction performance rather than cache performance. Thus, we intentionally set up experiments, which would exercise the instruction path of the memory system rather than the data path. We see there are fewer data cache misses than instruction cache misses even though the instruction caches are bigger.

Table 4.2: Instruction mix and cache misses for various benchmarks in percent of the total number of executed instructions

instruction	CMP	DES	FIR	Prime	Qsort	Minv	FFT	LU
nop	27.0	22.4	11.0	14.4	39.8	14.3	11.6	9.2
branch,jal	8.2	2.5	15.5	9.4	12.0	10.0	10.8	13.0
load	9.0	9.2	5.5	0	2.1	1.9	1.2	1.5
store	6.7	5.5	0.2	0	1.5	1.8	1.2	1.4
arith	26.2	29.2	38.7	35.8	16.6	32.1	33.2	32.5
logic	5.7	12.6	13.2	19.0	11.2	18.9	19.9	20.2
shift	6.7	8.7	0.5	8.1	3.5	7.5	7.3	5.8
movhi	0.2	1.7	0.4	0.8	1.8	2.3	2.2	2.3
cmps	9.9	7.63	4.0	11.0	11.9	8.9	8.4	5.9
predicated	2.3	6.7	12.7	6.1	0	6.3	7.9	10.5

4.5 Conclusions

This chapter introduced the GCC toolchain described the implemented a C compiler backend for Tinuso. We evaluated GCC's instruction scheduler for Tinuso's exposed 8 stage pipeline and we analyzed GCC's ability to exploit predicated execution. For a set of small C benchmarks Tinuso achieves an average speedup of 52% compared to a similar MicroBlaze configuration. The analysis the instruction mix of these benchmarks shows that still a high number of nop instructions are required to resolve hazards. Hence, we conclude that the optimization passes addressing delay slots are not adequate and we intend

as future work to either extend GCC's existing passes or, similarly to the c6x GCC backend, develop our own scheduling pass.

As only about 6% of the executed instructions make use of predicated execution we conclude that GCC's if-conversion pass is too limited for Tinuso. We intend to explore and implement our own more advanced pass based on current state-of-the-art [80, 92]. This should improve performance significantly. For example, Mahlke et.al report an average speedup of 72% using a fully predicated instruction set on their 8 issue processor [79].

The encoding of predicated execution in Tinuso is expensive as it takes up 4 bits of each instruction word and leads to a complex forwarding logic. However, Tinuso's predication scheme enables very efficient, though hand-coded, assembly code. Hand-coded assembly was used to implement a highly efficient low-level library, which give Tinuso a performance advantage.

Overall, GCC and the infrastructure it provides is a good choice for the Tinuso architecture. Describing instructions and architectural details is relatively straightforward and the quality of generated code is high. There are however several limitations as described above.

Chapter 5

On-Chip Interconnect

Tinuso is a processor architecture optimized for performance when implemented on FPGA. It applies superpipelining to exploit instruction level parallelism. Other techniques, such as superscalar or very long instruction word processors allow for executing multiple instructions in parallel but complex pipeline structures lead to a slower hardware implementation and thus, a lower instruction throughput. Therefore, Tinuso exploits parallelism at task level on multicore systems to improve performance. The logic integration of FPGAs has reached a point where large multi-processor designs can be composed. This means that efficient communication structures for FPGAs are required.

This chapter describes the design and implementation of a generic packet switched, 2D mesh on-chip network optimized for FPGA implementation. Section 5.1 introduces the on-chip interconnect and describes main design challenges. Related work on synthesizable interconnection networks is presented in Section 5.2. The architecture of the network and the router is introduced in Section 5.3. Section 5.4 describes the implementation of the network on the FPGA. The network is evaluated in Section 5.5. Finally, Section 5.6 summarizes and concludes this chapter.

5.1 Introduction

Tinuso applies superpipelining to attain a high system clock frequency and supports predicated execution to mitigate branch penalties. Superpipelining breaks pipeline stages into smaller stages and leverages instruction level parallelism by

executing operations partially overlapped. The current hardware implementation, the Tinuso I core, can be clocked as high as 376MHz on a Xilinx Virtex 6 device. Other techniques to increase instruction level parallelism such as superscalar or very long instruction word allow for executing multiple instructions in parallel. However, complex pipeline structures require a lot of hardware resources and limit the system clock frequency when implemented on FPGA, which leads to a lower instruction throughput. Hence, Tinuso applies parallelism at a higher level of abstraction to improve performance. Applications are split-up in a number of tasks that execute on an array of processor cores.

The evolution of FPGAs has reached a point where multiple processor cores, dedicated accelerators, and a large number of interfaces can be integrated on a single device. For example, Xilinx's Virtex 7 family comes with devices up to two million logic cells [118]. These devices allow for combining the processing power of hundreds of processor cores on a single FPGA. However, efficient communication structures optimized for FPGA implementation are required to successfully extract parallelism in multicore systems. Traditionally, point-to-point and bus based interconnects were used for system-on-chip, *SoC*, designs. These interconnects are simple to implement but do not scale well in systems with a large number of cores [73] or on FPGAs [64]. Instead, we argue for a 2D mesh network topology as it maps well to the FPGA fabric [57, 84]. Figure 5.1 shows a typical latency characteristic of an on-chip interconnection network. As long as few data packets are injected into the network, the latency to transport packets from source node to a destination node is low. As more packets are injected, contention increases, which leads to a higher network latency. Finally, at high injection rates the network is congested and latencies become unacceptable long.

In-order processor cores, such as Tinuso, typically stall while cache misses are resolved. Therefore, memory latency cannot be hidden, which makes cache misses expensive. Hence, the intercommunication network for Tinuso has to be optimized for a lowest possible latency and a high clock frequency to attain a high throughput. A plethora of synthesizable network designs exist. All apply design trade-offs to optimize the hardware design for different network properties such as throughput, latency, or a low hardware resource usage. The network for Tinuso is optimized for a high clock frequency and a lowest possible latency at low injection rates as marked with a circle in Figure 5.1. However, these properties are traded for additional hardware resources and higher latency at high injection rates.

This chapter describes the design and the implementation of Tinuso's communication structures and analyzes its scalability.

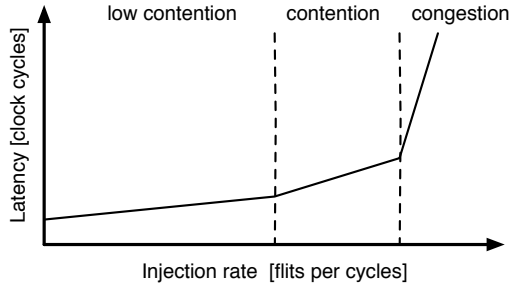


Figure 5.1: Latency vs. network traffic of a state-of-the-art interconnect

5.2 Related Work

Papamichael and Hoe developed a NoC generator, named CONNECT, that generates synthesizable RTL-level NoC designs [91].The network designs can be configured with a variable number of ports, virtual channels, flit width, buffer depth, flow control mechanisms and user defined routing schemes. This NoC generator is used to produce a multitude of designs and compare them with a state-of-the-art router for ASICs.

Due to FPGAs reconfigurability, FPGAs typically have a higher relative number of wires compared with logic and memory than ASICS. Because of different cost structures of FPGAs, NoCs optimized for ASICS may not be an optimal solution when synthesized on an FPGA. CONNECT designs are optimized for implementation on FPGA. The CONNECT router is a single stage pipeline that uses FPGAs distributed RAM to implement flit buffers and look-up tables for routing tables.

In experimental results they show that their 4x4 mesh configuration reduces the hardware resources by 58% over the Open Source Network-on-Chip Router RTL from Stanford University that is synthesized for FPGA [48]. Alternatively, they reach a 3-4x better performance for approximately the same hardware resource usage as the baseline implementation. They reach a maximum clock frequency of about 180 MHz for their 32-bit mesh implementations on Xilinx Virtex 6 devices. They conclude that designs with wider channels, fewer buffers, and less pipelined designs lead to a higher performance when implemented on an FPGA.

Huan and DeHon introduce a packet-switched NoC using split and merge primitives and compare the design with the CONNECT 2D mesh implemen-

tation [54]. Unlike CONNECT routers that come with a single pipeline stage only, they take a different design approach and construct routers with split and merge and queue primitives. The queue primitives use the Xilinx FPGAs built-in shift registers to implement buffers. The advantage of this design is that it allows for composing networks with different topologies and the decentralized implementation of routing and arbitration greatly reduces the logic complexity for each primitive. Moreover, routers can be pipelined at the level of the primitives. They show that adding a second pipeline stage significantly reduces the wiring delays that allows them to clock the design at clock frequencies of up to 300 MHz on Xilinx Virtex 6 devices. Just as CONNECT does, they use a Dimension Ordered Routing scheme, *DOR*, and a backpressure flow control mechanism. Experimental results show that the proposed split-merge NoC is up to 2-3 times faster than an equivalent CONNECT implementation while consuming about 13-37% more hardware resources. However, each additional pipeline stage introduces one additional clock cycle latency. In a multicore system where data packets have to pass multiple routers, this adds a significant latency and makes for example memory accesses more costly.

DESA is a distributed elastic switch architecture presented by Roca et. al. [100]. The interconnect is composed of a collection of independent switching modules, called AC modules. These switching modules are able to store, arbitrate and forward incoming flits. A simple lookahead routing algorithm is used and a backpressure mechanism pauses the transmission of stalled packets. The design does not have any centralized input buffer. Instead they distribute the buffers to the output registers in order not to use memory resources on the FPGA, such as distributed RAM or block RAM. The fine-grained decomposition of network in AC switching modules has two benefits. Firstly, this architecture allows for composing various network topologies. Secondly, the authors mention that the proposed architecture leads to better mapping on the FPGA because a maximum of flexibility is granted to the CAD tools. The design is evaluated by comparing to a mesh interconnect that consists of routers with a single pipeline stage. The evaluation shows that the DESA design increases the clock frequency by 50% while consuming about the same amount of resources as the baseline implementation. Nevertheless, the increased clock frequency comes at the cost of one additional pipeline stage, which increases the latency by one clock cycle per network hop.

LiPaR is a store-and-forward router optimized for a low hardware utilization on FPGA. Block RAMs are used to implement the packet buffers [106]. It is built for 2D mesh networks and supports five simultaneous connections. A XY-routing scheme and a store-and-forward flow control mechanism are used to

keep the decode logic simple and ensure an area-efficient implementation. The latency per hop of a store-and-forward router is proportional to packet size. The packet size may vary between 16 and 128 bits, which makes communication over a large network very costly. The router supports round-robin arbitration and utilizes 772 4-input LUTs and 10 block RAMs on a Xilinx Virtex 2 device. A 3x3 2D mesh network can be clocked at 32.25 MHz. Moreover, the presented router architecture also comes with multicast functionality [107].

MoCReS is a router architecture that applies virtual cut-through flow-control to transfer packets across multiple clock domains [57]. The router uses a FIFO buffer, implemented as block RAM, to transfer data across clock domains. It allows for operating multiple routers on independent clock frequencies and prevents the slowest router from restricting the operating frequency of the network. The use of multiple clock domains is of particular interest for heterogeneous systems on chips where a multitude of processing elements operate at different clock frequencies. The basic router design operates at clock frequency as high as 357MHz on a Xilinx Virtex 4 device. However, the performance of this network design is limited by a latency of at least 7 cycles per hop.

Lu et. al. introduce a generic router architecture optimized for FPGA implementation [78]. It supports network topologies such as ring, 2D mesh, 3D cube, and hybrid configurations. A low latency of 2 clock cycles per hop is achieved by using wormhole switching. The router is pipelined to enable high frequency system clocks. A credit based flow control mechanism is used. A flit can only be transmitted when buffer space is available. The packet header contains additional auxiliary routing information to support a look-ahead routing scheme. The look-ahead routing logic computes the output of the next router in the path and thereby simplifies the decode logic. The router supports arbitration schemes such as round-robin and least-recently-served and uses buffers implemented as block RAMs. A router with five ports and a 32 bit data-path width utilizes about 700 LUTs and can be clocked as high as 220MHz on an Altera Stratix III device.

5.3 Architecture

On cache misses, the Tinuso pipeline stalls. Therefore memory latency cannot be hidden, which makes cache misses expensive. Hence, the performance of a Tinuso core highly depends on the latency in the network. We therefore aim for a network with low latency and high data rate when implemented on FPGAs. Typically, an FPGA consists of a two-dimensional array of logic elements called

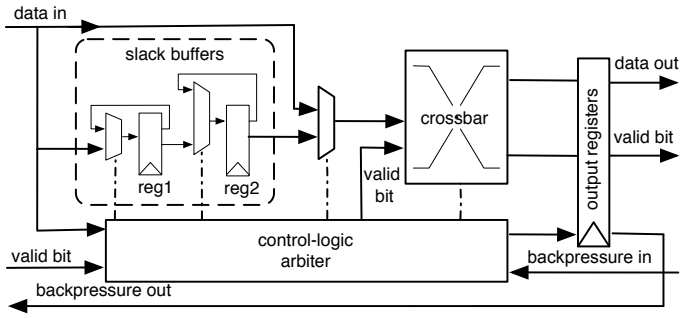


Figure 5.2: Block diagram of a wormhole router with pipelined backpressure flow-control

configurable logic blocks, *CLBs*, that are interconnected by horizontal and vertical routing channels. We argue for a 2D mesh network topology as it maps well to the FPGA architecture. The network consists of a number of routers that are connected through unidirectional links. A router has five bidirectional ports, namely, North (N), East (E), South (S), West (W), and Home (H). We have implemented a wormhole router with a backpressure flow-control mechanism and only use flip-flops at the output to attain a latency of one cycle per hop.

We decided for an XY routing scheme because it is deterministic, deadlock-free and very simple to implement. Figure 5.2 shows the block diagram of the router. It consists of crossbar, switch arbiter, slack buffers, output registers, and a control unit implemented as a finite state machine, *FSM*. Data packets are broken into a sequence of flow control digits, *flits*. The network supports packets that consist of a header flit and an arbitrary number of data flits. To keep the communication overhead low, all routing information is encoded in a single header flit. Each data link includes a status signal that indicates whether the data is valid or not. Two successive packets must be separated by at least one invalid flit. This allows the router to detect the header and tail flits of a packet.

The finite state machine of an input router port is shown in Figure 5.3. It remains in the idle state until a packet arrives. Then the destination node address is extracted from the header flit. An output port is selected following the routing scheme and it is checked whether the desired port is available or not. If more than one flit arrives at the same time and wants to use the same output

resource, the arbiter decides which one will proceed. If the desired output port is available and the header flit got the permission from the arbiter to proceed, it is stored in the output register and the router moves to the transmit state. In cases where the desired output port is not available or the arbiter prioritizes another packet, the flit is retained in slack buffers and the backpressure signal is set. The router input port remains in the arbiter state until the desired output resource is available again. The backpressure signal propagates up-stream to stall the data flits until the desired output port is available. Hence, routers that hold the data flits of the packet will remain in the retransmit state until the backpressure signal is released. The network supports both a fixed priority and a round-robin arbitration scheme, which is set when the design is synthesized:

- The round-robin arbitration scheme is implemented with a small queue in the arbiter where the highest scheduling priority is given to the packet that has waited the longest time.
- The fixed priority scheme is designed for low hardware costs and to reduce contention in the network. Packets that are routed in the second dimension (Y direction) are given a higher priority than packets routed in the first dimension (X direction) because they are likely closer to their destination. Moreover, packets from the home node are given the lowest priority. Thus, the arbitration scheme limits the number of packets in the network by prioritizing packets that are in the network already. Successive packets are separated by at least one invalid flit. This allows stalled packets of a port with a low priority to proceed before the next packet at the port with the high priority arrives. The arbitration scheme provides some fairness but is not entirely fair.

The arbiter state is not time-critical, hence it is possible to implement an arbitration scheme that prioritizes packets of a certain type. For example, memory request messages can be routed with a higher priority than other messages.

5.4 Implementation

We implement the router architecture described in Section 5.3 in VHDL. We decided for a behavior-level description as it is shorter, easier to adapt, and less error prone than descriptions at a lower level. Moreover, behavioral-level designs are independent of the technology and can easily be migrated to alternative platforms.

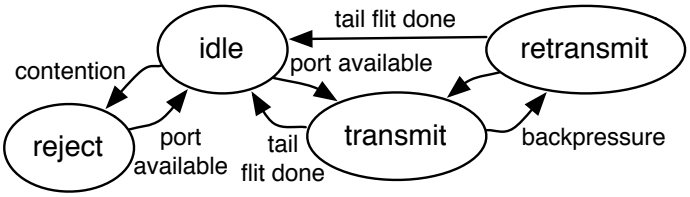


Figure 5.3: State diagram of router input port FSM

We identified the time-critical path of the design in the decode logic in the idle state. The decode logic extracts the destination address of the header flit and determines an output port according to the XY routing scheme. If the header flit only contains the coordinates of the destination node, costly comparison operations are necessary to determine the route of the packet. Therefore, we apply look-ahead routing to simplify the time-critical path by computing the output port in the previous network hops. However, look-ahead routing comes at the cost of additional complexity in the network interface where auxiliary routing information is computed and additional bits in the header flit are occupied. When the home node injects a packet, it needs to determine, which output direction to take. Following the XY routing scheme the next routers in the path need to redirect the packet at most once in Y direction. Hence, we use two bits to encode each routing dimension. These bits are set when a message is composed in the network interface. An additional bit specifies, which routing dimension is active. This bit is updated by the routers along the path. Table 5.1 provides an overview over the encoding of auxiliary routing information in the packet header.

The proposed router architecture requires the ability to retransmit flits if contention occurs. We use a backpressure flow control mechanism to manage contention. It was necessary to pipeline the backpressure feedback loop to reduce the routing delay in the time-critical path of the design.

As a consequence, a backpressure signal that reports contention arrives with a delay of two clock cycles. Hence, the slack buffers store the last two transmitted flits and the router re-transmits them when the backpressure signal is released. This is done in the retransmit state, which is not time-critical. While this leads to a higher system clock frequency, it slightly increases the network latency when contention occurs because data needs to be re-transmitted.

The data link-width is configurable. However, it must not be smaller than the minimal header size, which includes address bits for X and Y coordinates of

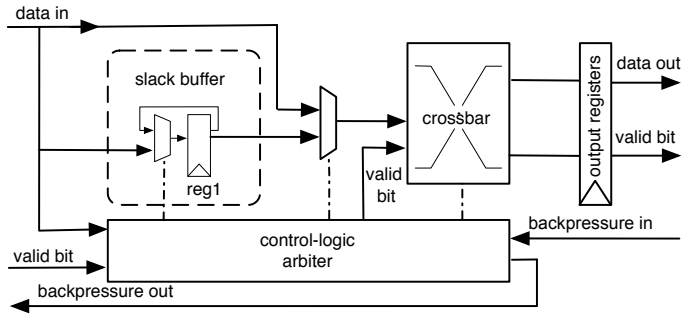


Figure 5.4: Block diagram of a pipelined wormhole router with backpressure flow-control

the destination node and 5 bits of auxiliary routing information. Other networks that are optimized for FPGA implementation typically use buffers implemented as block RAMs [57, 78, 106]. We do not use block RAMs because the output delay of block RAMs would slow down the design or lead to a pipelined router design with an additional clock cycle latency. Moreover and importantly, block RAMs also restrict the placement when the network is mapped to the FPGA. Instead, we use the flip-flops of the CLBs to implement buffers.

5.4.1 Packet Definition

The network is designed to support a configurable link-width. To avoid alignment problems, the link-width must be a multiple of 16 bit. Figure 5.5 shows packet headers for various link-widths, which encode of the following items:

- X/Y Coordinates of the destination node. There are 5 bits reserved for each dimension, which allows for network configurations with up to 1024 nodes.
- 6 bits are used to encode auxiliary routing information to implement lookahead routing.
- An identifier is used to categorize messages. The category defines the organization of a given message. In systems with a link-width of at least 32 bits, the identifier is included in the first flit. Thus, a cache controller can trigger appropriate actions already after reading the first flit.

Table 5.1: Auxiliary routing information in packet header

Bit	Value	Description
5,4		direction of first routing dimension
	"00"	north
	"01"	east
	"10"	south
	"11"	west
3,2,1		direction of second routing dimension
	"001"	no second routing dimension
	"010"	west
	"100"	east
0		routing status
	"00"	routing packet in first dimension
	"00"	routing packet in second dimension

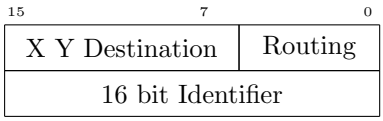
When the pipeline detects a cache miss, the cache controller implicitly generates a memory request message that is sent to the memory controller. Figure 5.6 defines memory request messages that consist of two packet headers, a memory address and an arbitrary amount of spare data, which can for example be used for a Cyclic Redundancy Check, *CRC*, to detect transmission errors. The first packet header describes the route from a processor to the memory controller. The second header describes the route from the memory controller back to the processor core that initiated the memory request.

Some cache misses require to write modified data back to main memory. The cache controller then composes first a memory write-back message as defined by Figure 5.7, followed by a memory request message. The XY routing scheme ensures that the memory controller receives first the write-back message before the memory request message. This is important to maintain memory consistency.

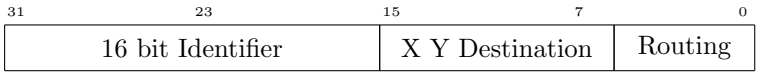
5.5 Results

To evaluate the proposed network we derive clock speed and hardware resources of various router configurations and measure the network latency of random traffic. Table 5.2 lists hardware resources and maximum clock speed of a single router implementation on various Xilinx FPGA families. The results are based

- 16-bit Packet Header:



- 32-bit Packet Header:



- n x 16 bit Packet Header:

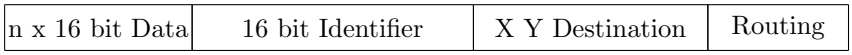


Figure 5.5: Packet headers of various link-width

on Xilinx ISE 14.6 "place and route reports". We conclude that the design scales well with the data link-width. For example, a router with a 16 bit link-width consumes about 60% of the hardware resources of a design with 32 bit link-width. The price to implement the round-robin arbitration scheme is relatively small as it requires few additional LUTs only and does not affect the maximum clock speed.

To evaluate the network latency we compose two multicore setups consisting of 3x3 and 4x4 nodes and a bit link-width of 32 bits. Figure 5.8 illustrates the test setup where each network node is equipped with a state machine that

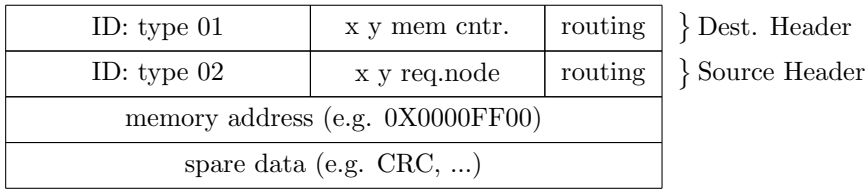


Figure 5.6: Memory request message

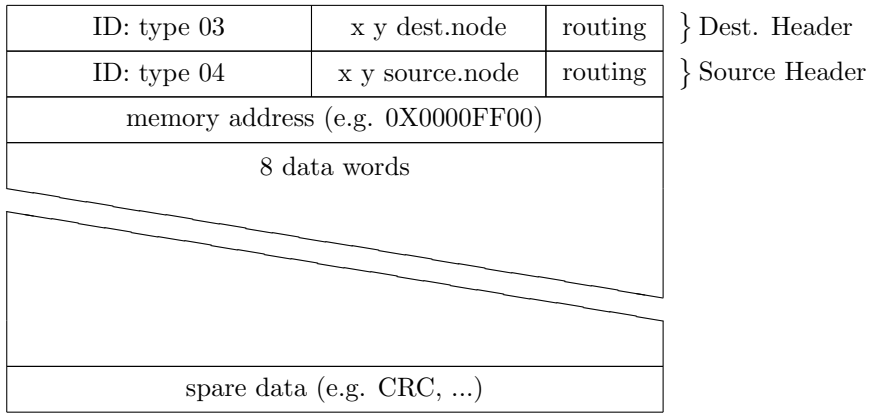


Figure 5.7: Memory write-back message

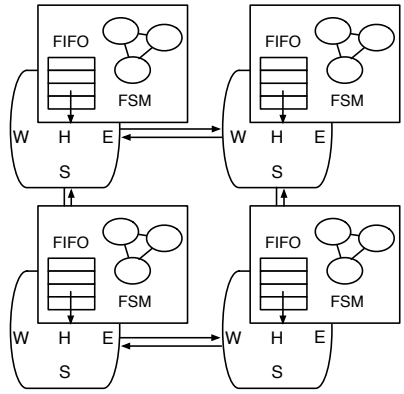
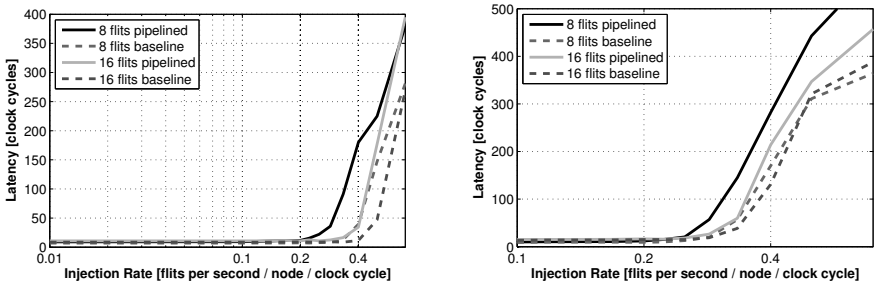


Figure 5.8: Overview of the test setup to measure the network latency

generates random data packets and stores them in a FIFO buffer. All state machines generate the same number of packets and use a uniform random distribution to generate the data packets. The packets in the FIFO buffer are then injected into the network. We use a VHDL simulator to model the complete network implementation and measure the average network latency of random

FPGA Family	Arbitr. Scheme	16 bit link LUTs	16 bit link MHz	24 bit link LUTs	24 bit link MHz	32 bit link LUTs	32 bit link MHz
Virtex 7	r-robin	683	437	913	428	1121	394
Virtex 7	fixed	659	437	878	428	1077	394
Virtex 6	r-robin	679	412	903	372	1097	347
Virtex 6	fixed	660	412	880	372	1052	347
Spartan 6	r-robin	753	223	1031	208	1299	204
Spartan 6	fixed	686	223	949	208	1204	204

Table 5.2: Overview of hardware resource usage and clock frequency of various router configurations on Xilinx FPGAs of speed grade -3.



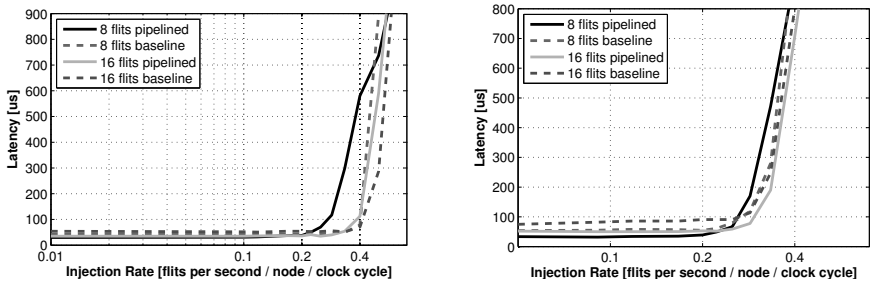
(a) 3x3 2D mesh

(b) 4x4 2D mesh

Figure 5.9: Average latency in clock cycles vs. packet injection rate of a 3x3 and a 4x4 2D mesh network.

packets for various injection rates. We define network latency as the number of clock cycles starting from the cycle a header flit is stored in the FIFO buffer until the packet has reached its destination.

The injection rate corresponds to the number of packets that are stored in all FIFO buffers per clock cycle. We run experiments with a fixed packet size of eight and sixteen flits. To get unbiased results, we warm the network and run up to 50 iterations per test point. We compare the latency of a Tinuso network with a baseline router implementation without pipelined feedback loop to manage contention. The baseline router design is illustrated in Figure 5.4. It is simpler as it only requires a single slack buffer. However, the backpressure



(a) 3x3 2D mesh

(b) 4x4 2D mesh

Figure 5.10: Absolute average latency in μs vs. packet injection rate of a 3x3 and a 4x4 2D mesh network.

feedback loop restricts the clock frequency of the design. Figure 5.9 shows the average latencies for various injection rates. The latency of the baseline implementation is lower when contention occurs because of the smaller slack buffer fewer flits need to be re-transmitted. We observe a very low latency up to an injection rate of 0.3 flits per node per cycle. At injection rates higher than 0.4 flits per node per cycle the network is congested and latencies become unacceptable long.

In Figure 5.10 we derive the absolute latency in μs. We scale the latency with the maximum clock frequency. On a network with 16 nodes we measure a maximum clock frequency of 304 MHz for the Tinuso router while the setup with the baseline router implementation can be clocked at 164 MHz only. The maximum frequency of the multicore system is lower than the clock speed of a single router implementation because it includes the routing delay of the data links between the routers. The Tinuso network performs better at low injection rates but there are situations at high contention where the latency in the network with the baseline router is lower.

5.6 Conclusions

This chapter described the design, implementation and evaluation of communication structures for Tinuso multicore systems. We argued for a 2D mesh network topology as it maps well to the structure of FPGAs. The proposed

router architecture uses wormhole switching and a backpressure flow control mechanism to attain a latency of one clock cycle per hop. Routing scheme and flow control mechanism were optimized for high system clock frequency. A deterministic XY, lookahead routing algorithm was applied to simplify the decode logic in the routers. Moreover, the feedback loop to manage contention was pipelined to reduce the routing delay of the time critical path in the design.

The network latency of random traffic is measured and compared to a baseline router implementation. We measure very low latencies for injection rates up to 0.3 flits per node per cycle. We show that a pipelined feedback loop to manage contention leads to a significantly higher clock speed and a lower network latency at low injection rates while consuming moderately more hardware resources. We conclude that our behavioral-level network implementation delivers a high performance, is scalable, and allows for composing large multicore systems.

Chapter 6

Multicore Simulation Platform

Multicore systems have the potential to improve performance, energy, and cost properties of embedded systems but also require new design methods and tools to take advantage of new architectures. Due to the limited accuracy and performance of pure software simulators cycle accurate hardware simulation platforms are required for research on embedded multicore systems.

Tinuso processor pipeline and interconnection network are designed with the perspective of being used in large multicore systems. The design comes with both instruction and data caches and a cache controller, which implicitly generates and processes memory requests. However, if all communication between processor cores has to go through main memory, we obtain a high contention rate in the interconnection network, which limits the scalability of the system. Alternatively, extending Tinuso with infrastructure for direct communication between processor cores offloads main memory and improves the scalability.

As the hardware design is easily extendable and optimized for a high performance when implemented on an FPGA, Tinuso is an attractive hardware platform for research on multicore systems and parallel programming.

This chapter describes the design and implementation of a communication interface for efficient communication between processor cores and evaluates the hardware scalability of the system. Section 6.3 describes the design and implementation of the communication interface. Tinuso system components and interfaces are described in Section 6.4. Section 6.5 evaluates the scalability of

Tinuso multicore systems. Finally, Section 6.6 summarizes and concludes this chapter.

6.1 Introduction

Parallel computing and multicore architectures have spread into most areas of digital system designs. While it is conceptually trivial to envision how such systems are designed by composing processor cores and an interconnection network, the programming of such a multicore system represents a huge challenge. New approaches for programming languages, programming models, and runtime systems are required to support the programmer in expressing parallelism and to efficiently execute parallel programs. To exploit the full potential of such multicore systems, the hard- and software development has to go hand-in-hand. However, software simulators do not cover all aspects of multicore systems, suffer from insufficient accuracy and have a limited throughput when simulating with a large number of processors [116].

We therefore propose the use of Tinuso multicore systems as simulation platform to capture the behavior of a high number of processor cores, network interfaces, and the corresponding interconnect at a high clock rate. Multiprocessor systems can roughly be classified in two categories. First, there are shared memory multiprocessors systems, which communicate through a common memory. Second, message passing systems do not have a shared memory address space and communicate by sending messages directly from one processor core to another. Tinuso is a multicore system where all processor cores may share a common memory address space. However, if all communication between processor cores has to go through main memory we may end up in situations in which communication data rates lead to a high contention rate in the interconnection network or exceed the memory bandwidth.

As Tinuso processor pipelines execute instructions in-order and stalls on cache misses it is difficult to hide memory latency. Hence, communication between processors through shared memory may become a major performance bottleneck and limits the scalability of the system. Therefore, direct communication between processors must be enabled for Tinuso multicore systems. It offloads main memory and leads to communication patterns that make better use of the 2D mesh interconnection network. A communication interface that allows for efficient communication between processors must be tightly integrated in the processor cores to keep the overhead for composing messages low. Tinuso comes with instructions to access machine specific registers as described in

Appendix B. Hence, these registers can be used to connect the communication interface with the processor pipeline. A high performance and rich toolchain support are important factors for the success of a simulation platform for research on multicore systems. Moreover, a simulation platform must be easily extendable and adaptable.

All Tinuso system components are therefore implemented as hierarchical behavior-level descriptions in VHDL. Behavior-level designs typically are shorter, easier to adapt, and less error prone than descriptions at a lower level and can easily be migrated to alternative platforms. Tinuso multicore system and toolchain provide the ability for customizations at all levels. For example, it is possible to add processor cores, define special instructions, and change the interconnect link-width.

The hierarchical implementation allows for an easy substitution of interconnection networks to explore alternative network topologies or changes in the memory hierarchy. Moreover, Tinuso systems can be equipped with performance counters that capture the instruction mix of an application, monitor cache misses and network traffic. Therefore, Tinuso multicore systems are an attractive choice for research on multicore systems and parallel programming. Finally, Tinuso multicore systems of various size are composed to evaluate the scalability on current state-of-the-art high-end FPGAs.

6.2 Related Work

ProtoFlex [29] is an FPGA-accelerated cycle accurate hybrid functional simulator designed for large-scale multiprocessor hardware and software research. ProtoFlex provides the ability to run stock commercial operating systems with I/O support. Its transplant technology uses FPGAs to dynamically accelerate only common simulation cases while relegating infrequent, complex corner cases to software simulation. By working in concert with existing full-system simulators such as Flexus [27], transplanting avoids the costly construction of the entire target system in FPGA. Large scale multicore systems can be simulated by time-multiplexed emulation where multiple processors in the target model are mapped to a smaller number of physical units [28]. Protoflex uses interleaved in-order pipelines without data forwarding or stalls to keep the hardware simple and to ensure multithreading, which is required by the engine to run multiple processor contexts. ProtoFlex is currently targeting SPARC V9 platforms.

RAMP [115] is a collection of researchers and projects that work on an FPGA-based emulator of parallel architectures. The RAMP vision provides

hardware platforms that allow software developers to start their development on innovative architectures and programming models. For instance, Ramp BLUE efficiently emulates a prototyped distributed-memory message-passing architecture [65]. As only 12 MicroBlaze cores fit into a Virtex-II Pro FPGA, a total of 84 devices are required to build a system with 1008 MicroBlaze cores. RAMP implements a generic crossbar for on-chip communication and makes use of the FPGA's fast serial off-chip links to form a system-level 3D mesh network. RAMP is a message passing architecture with a static memory partitioning where each core has a total of 256 MB private memory. Tinuso comes with a global address space, which can be partitioned into private memory sections, if required.

Both ProtoFlex and RAMP simulation platforms use off-the-shelf processors with the intention to ease the platform access with a large number of software components available. Oppositely, we aim for a processor that is easy extendable to maintain the flexibility required for the research on the hardware/software interface for programming model support of multicore systems.

US-FAST [24] describes another form of hybrid simulation that uses FPGAs to accelerate cycle accurate simulation. It is capable of efficiently simulating general-purpose speculative processors as well as multiprocessors. The simulation problem is partitioned into two levels to face the complexity of such systems. A functional model is responsible for simulating at the instruction set architecture ISA and functional peripheral level, while a timing model implemented in the FPGA is used for modeling micro-architectural structures that impact timing. Multicore aspects such as packet-switched interconnects are not covered by this simulator.

6.3 Implementation of the Communication Interface

Figure 6.1 shows the internals of the Tinuso network interface. Its primary task is to connect a processor core to a router. Cache misses are translated into memory request messages. Outgoing messages are placed in a buffer and sent out as soon as network resources are available. The pipeline stalls while cache misses are resolved. Hence, the pipeline does not access the cache. Therefore, incoming memory data is written directly into the caches without an input buffer.

The dotted frame in Figure 6.1 illustrates the communication interface.

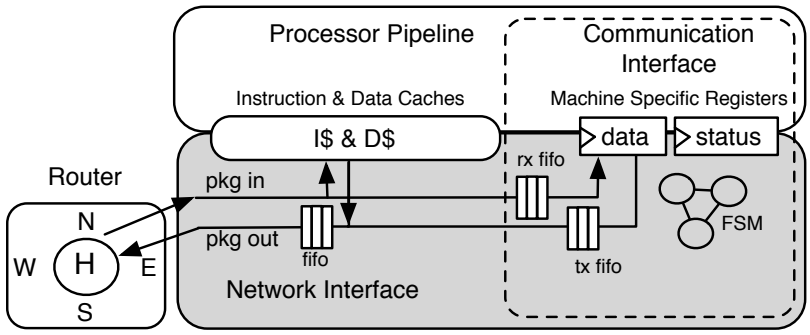


Figure 6.1: Implementation of the communication interface

While the cache controller implicitly generates and processes memory requests, the communication interface allows for explicit communication with other processing elements. We define network messages as explicitly generated messages composed by processor pipeline and communication interface. The pipeline accesses network messages by reading and writing to machine specific registers. The communication interface requires a set of buffers for both incoming and outgoing messages to avoid conflicts with memory request messages. The communication interface is integrated in the cache controller and consists of message buffers and a control state machine that read and dispatches network messages and sets status bits. Message buffers are implemented as FIFO queues in block RAMs. There is a port for the network interface and one for processor pipeline.

Figure 6.2 illustrates how the processor pipeline reads and composes network messages. Incoming network messages are identified through an ID in the packet header and put into a buffer in the communication interface. Then the *packet ready* and the *packet header* bits are set. The pipeline periodically polls for incoming network messages by reading these status bits. Instructions that read special registers, *mfms*, are used to read status bits and packet data. Whenever a packet data is read by the processor core, the buffer index for incoming network messages is incremented to make the next data flit available. This process is repeated until complete network message is read. The end of a message is either detected when the *packet ready* status bit is reset or a new *packet header* bit is read.

The processor pipeline composes network messages by writing data to a specific register, which feeds data in the outgoing message queue. Instructions that

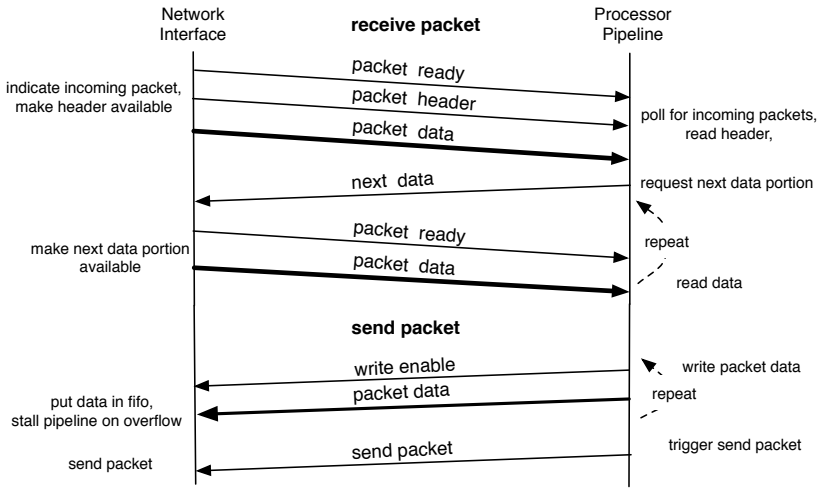


Figure 6.2: Communication interface

write special registers, `mtms`, are used for this purpose. Once the message is complete, the processor pipeline triggers the dispatch of the network message by setting the `send packet` status bit. The control state machine in the communication interface is able to detect overflow situations in the buffer for outgoing messages and can pause the creation of new messages. Overflow situations for the incoming messages are more difficult to handle since it is not possible to pause message creation in remote processor cores. The communication interface therefore drops incoming network messages if the input buffers are full and set an `overflow` status bit. It is then up to the software to recover from such situations.

6.4 Simulation Platform Components and Interfaces

Figure 6.3 illustrates the hierarchical design of a Tinuso processor multicore system, which includes processor pipeline, cache controller, network interface, router, and memory controller.

Processor Pipeline. The Tinuso processor core is a three operand, load-

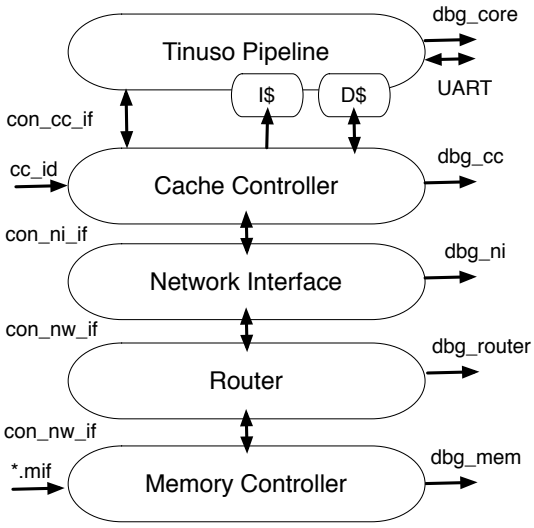


Figure 6.3: System interface overview

store architecture with a fixed instruction word length of 32-bits. The in-order pipeline has a total of 8 pipeline stages and is optimized for a high instruction throughput when implemented on an FPGA. The architecture, implementation and evaluation of the Tinuso processor core is described in Chapter 3.

Cache Controller. The cache controller has four main objectives. Firstly, it generates memory request messages when the processor pipeline observes cache misses. Secondly, it updates the cache based on incoming data packets from the network. Thirdly, it includes special registers that the pipeline can read and write with `mtms` and `mfms` instructions. This infrastructure is used to implement the communication interface and to read out performance counters. Finally, the cache controller has to update the status register for load linked, `ll`, and store conditional, `sc`, instructions. The cache controller is tightly connected to the pipeline and can set program counter, if necessary. Hence, it is able to stall the pipeline if required. The current implementation includes a finite state machine for both incoming and outgoing data. There is support only for a single processor pipeline with corresponding first level caches. The standard implementation does not support memory consistency in hardware. A suitable coherency mechanism for Tinuso multicore systems is described in Chapter 8.

Network Interface. The network interface is responsible to inject data packets into the interconnection network and to forward incoming data to the cache controller. Hence, it implements the backpressure flow-control mechanism and includes a fifo buffer to store data packets until they are injected into the network. Incoming data packets from the network are directly forwarded to the cache controller. Given the low complexity of the network interface, it would be possible to integrate this functionality into the cache controller. However, it is designed to work with a plethora of cache controllers. Beside the simple cache controller that connects to a single core and does not support cache coherency there will come up implementations of cache controllers for multiple processor cores with a shared level-two cache and support for cache coherency. Hence, it is easier to maintain the network interface when it is implemented in a separated component. Moreover, on a single core implementation there is no need for the functionality of the network interface.

Router. The five port router is designed to compose generic 2D-mesh on-chip networks. To attain a high data rate the XY routing scheme, flow control mechanism and arbitration scheme are optimized for high clock frequency and low latency. The router can be configured to use a fixed priority scheme or a round-robin arbitration scheme. The data link-width can be configured to match the needs of an application. However, it may not be smaller than 16 bits. The router architecture, implementation and evaluation is described in Chapter 5.

Memory Controller. The memory controller is currently only able to model SRAM. However, this is sufficient for most applications as the clock frequency of the FPGA multicore system is lower than the data rate of modern SDRAM memory blocks. The memory controller is directly connected to the interconnection network. The design is synthesizable but is not suited for hardware implementation because memory is implemented as a large array that does not fit into the FPGA's block rams.

There is a fifo buffer at the input of the memory controller pipeline to store incoming memory requests and write backs. Memory is implemented as a huge array that is initialized with a *.mif file. Hence, there is no bootloader required. The Tinuso toolchain includes a script to convert *.elf executable into *.mif files. At the output of the memory pipeline there is a fifo buffer where output data is stored until network resources become available. The memory controller implements the backpressure flow-control mechanism to ensure that data is injected correctly into the interconnection network. The simple architecture is designed to run efficiently in the hardware simulation environment.

Simulation Environment. The Tinuso VHDL sources are architecture

independent. Therefore, all Tinuso components can be simulated with a broad range of simulation tools. We primarily used GHDL, an open source VHDL simulator [46] for functional simulation. GHDL does compile VHDL code directly to machine code rather than synthesizing it. Moreover, GHDL allows for including subprograms written in programming languages such as C or Ada and is therefore well suited for hardware software co-simulation. Figure 6.3 shows how to connect the system components to compose a multicore system. Each component of the Tinuso architecture comes with a set of interfaces that allow for an easy composition of multicore systems. Each interface consists of two records of signals: One for incoming data and one for outgoing data. Grouping signals to a record in VHDL has two advantages. Firstly, the higher abstraction allows connecting components with a single signal. Secondly, it is possible to add or remove signals from the record without all files that use a connection of this type need to be updated. Each component has a debug record that can be used within the simulation environment to log the values of all registers of a component. However, this is a huge data record that slows down the simulation and it needs to be disabled when the design is synthesized for hardware implementation as it exceeds the I/O capabilities of most FPGA devices.

Table 6.1 shows the interface between the processor pipeline and the cache controller. There are control signals that allow the cache controller to reset or halt the pipeline. It is necessary to prevent the pipeline from accessing the caches while the cache controller updates the caches. The cache controller can therefore stall the pipeline if necessary. This is relevant for multicore systems when a hardware cache coherency mechanism is implemented. The cache controller may be triggered by an event from a remote processor to invalidate or write back portions of the cache data without interfering with the program execution in the local processor pipeline. The cache controller interface also includes control signals for instruction and data cache misses. For example, when the pipeline observes a data cache miss, it asserts the `data_cache_miss` signal, flushes the pipeline and goes into reset mode. The cache controller checks whether data needs to be written back and triggers a memory request. Once it has received the requested memory and the cache is updated it asserts the `data_cache_miss_done` signal to restart the pipeline. The cache controller has a write port to the instruction cache and a read and a write port to the data cache. Simultaneous read and write operation on the data cache is permitted.

The interface between network interface and cache controller is defined in Table 6.2. Incoming data packets to the network interface are directly forwarded to the cache controller. It is up to the cache controller to decode the data packages. The backpressure flow control mechanism is used to indicate congestion in

Table 6.1: Interface between cache controller and processor pipeline

Cache Controller			Processor Pipeline	
→	core_id	[8 bit]	→	pipeline control
→	reset	[bool]	→	
→	reset_pc	[30 bit]	→	
←	current_pc	[30 bit]	←	
←	instr_cache_miss	[bool]	←	instruction cache miss
←	instr_cache_miss_addr	[30 bit]	←	
→	instr_cache_miss_done	[bool]	→	
←	data_cache_miss	[bool]	←	data cache miss
←	data_cache_miss_addr	[30 bit]	←	
→	data_cache_miss_done	[bool]	→	
→	instr_cache_w_en	[bool]	→	instruction cache write
→	instr_cache_w_addr	[10 bit]	→	
→	instr_cache_w_data	[31 bit]	→	
→	instr_cache_w_tag	[21 bit]	→	
→	data_cache_r_addr	[10 bit]	→	data cache write
←	data_cache_r_data	[32 bit]	←	
←	data_cache_r_tag	[20 bit]	←	
→	data_cache_w_en	[bool]	→	data cache read
→	data_cache_w_addr	[10 bit]	→	
→	data_cache_w_data	[31 bit]	→	
→	data_cache_w_tag	[21 bit]	→	
→	com_if_packet_ready	[bool]	→	message passing interface
→	com_if_packet_header	[bool]	→	
→	com_if_read_data	[32 bit]	→	
←	com_if_write_en	[bool]	←	
←	com_if_write_data	[32 bit]	←	
←	com_if_send_packet	[bool]	←	
←	com_if_read_en	[bool]	←	

Table 6.2: Interface between network interface and cache controller

Network Interface			Cache Controller	
→	data	[≥ 16 bit]	→	data from network
→	data_valid	[bool]	→	
←	data	[≥ 16 bit]	←	data to network
←	header	[bool]	←	
←	tail	[bool]	←	
←	write_en	[bool]	←	

Table 6.3: Interface between router ports, network interface, or memory controller

Router Ports (N, S, E, W, H), Network Interface, Memory Controller			Router Ports (N, S, E, W, H) Network Interface, Memory Controller	
→	data	[≥ 16 bit]	→	out data
→	data_valid	[bool]	→	
→	backpressure	[bool]	→	
←	data	[≥ 16 bit]	←	in data
←	data_valid	[bool]	←	
←	backpressure	[bool]	←	

the network. The interconnection network ensures that incoming data packets consist of a continuous stream of flits and that data packets are separated by at least one invalid flit. Hence, a single `data_valid` control signal is sufficient to determine header and tail flit of a data packet. As there is no fifo for incoming data, the cache controller must treat incoming data from the network with the highest priority. Control signals to mark the header flit and the tail flit of a package are required to ensure that the network interface is able to identify start and end of a data packet.

Table 6.3 defines the interface to the interconnection network. As both memory controller and network interface connect to the interconnection network they use the same interface. For each direction there is a data path with a configurable link width and corresponding control signals. These control signals indicate whether data is valid or not and if backpressure flow control is active or not. A router typically has five of these bidirectional interfaces.

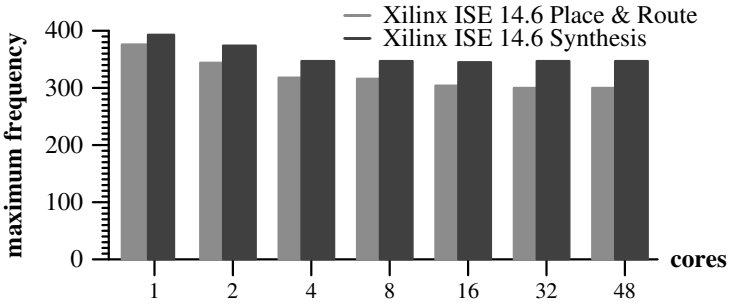


Figure 6.4: Hardware scaling of Tinuso multicore systems

6.5 Scalability of Tinuso Multicore Systems

To evaluate the scalability of Tinuso multicore systems we compose Tinuso multicore systems with a various number of cores and derive the maximum clock frequency of the system on a Virtex 7 device. We populate the network nodes with Tinuso processor cores. Both network and processor cores operate in the same clock domain. The processor cores include communication interfaces and instruction and data caches with a size of 4 Kbyte each.

Figure 6.4 shows the results after the synthesis and the placed and routed design. With increasing system size it becomes more difficult for the tools to map the design on the FPGA fabric. The single core configuration does not include a router and is therefore faster than the other designs. Designs with more than 4 processor cores require floor-planning in Xilinx PlanAhead to attain an acceptable clock speed. We define local area constraints (Pblocks) and assign the processor cores to them.

We do not constrain the placement of the network and let the Xilinx toolchain find a performance-optimized placement. However, for very large systems with more than 64 processor cores the tools report a high system frequency after synthesis but the tools are not able to map the design efficiently and report slow place and route results. Xilinx's Virtex 7 family comes with devices up to two million logic cells that allow for Tinuso multicore configurations of up to 480 processor cores. Hence, we currently explore how to automatically generate fine-grained design constraints to efficiently support the placement for very large systems.

6.6 Conclusions

This chapter described the design and implementation of a communication interface for Tinuso multicore systems. It allows for sending and receiving network messages. It is designed with the aim to offload memory traffic. With the advent of this communication interface, processor cores in a multicore system can either communicate directly or through shared memory. Tinuso multicore systems can therefore be used for research on multicore systems and parallel programming.

The hierarchical design of Tinuso system components with simple interfaces allow for easily composing and adapting multicore setups. Tinuso multicore system and toolchain allow for customizations at all levels. For example, it is possible to add processor cores, define special instructions, and change the interconnect link-width. Tinuso multicore systems can either be simulated with any VHDL simulation tool, which is useful for low level debugging, or alternatively the system can run on hardware to attain a high simulation speed.

However, a high level of detail comes at the cost of simulation speed. A standard off-the-shelve desktop machine attains a simulation speed of approximately 300Hz for a Tinuso multicore configuration with four processor cores with all debugging output enabled.

As research platforms with a very high number of processor cores are desirable, the hardware scalability of Tinuso multicore systems was evaluated. The system scales well up a system size of 48 cores. We reached a maximum clock frequency of 300 MHz on a Xilinx Virtex 7 device, which corresponds to a peak switching data rate of 9.6 Gbits/s per link. The Xilinx tools have problems to map larger multicore systems on the FPGA. Substantial floor-planning support is required to attain a high system clock speed. We conclude that the Tinuso behavioral-level implementation delivers a high performance, is scalable, allows for composing large multicore systems and is therefore an attractive multicore research platform.

Chapter 7

Tinuso Multicore for Synthetic Aperture Radar Data Processing

When system designers have to decide for an embedded computing platform, they need to evaluate a plethora of system properties such as development time and costs, computing performance, unit costs, power, reliability, and mechanical properties. Discrete embedded processors and systems on chips are typically cheap and come with a high number of interfaces and rich tool chain support. In recent years several large multicore platforms have entered the market. However, this market is very dynamic and it is not clear which architectures will still be supported in a few years. GPU's provide a huge computing power per price unit but these systems require a host system, are power hungry and the GPU programming model may not fit to the application.

Custom hardware solutions can be optimized for a given application. ASIC technology typically provides the highest performance, lowest power consumption and lowest unit cost. However, the high setup and development costs make ASICs only economically viable for high volume production. Alternatively, FPGAs have a higher unit cost but no setup costs. The logic integration of FPGAs has reached a point where multiple processor cores can be integrated in a single device.

This chapter presents a case study where a SAR data processing application is successfully mapped to a multicore system on an FPGA. The study analyzes

the parallelization of the data processing algorithm, application specific modifications to the existing Tinuso infrastructure, and the application mapping to the hardware. This chapter is based on a publication and is structured in the following sections [104].

Section 7.1 introduces the case study. The radar application and how to parallelize the algorithm is described in Section 7.2. Related work is discussed in Section 7.3. Section 7.4 describes the system architecture, which includes all application specific modifications on the Tinuso processor core. Hardware organizations and scalability of the system are discussed in Section 7.5. Results such as computing performance, clock speed, utilized hardware resources, and network traffic are presented in Section 7.6.

7.1 Introduction

Synthetic aperture radar, *SAR*, is a form of imaging radar that provides high quality mapping independent of light and weather conditions. SAR is used across a wide range of scientific and military applications including environmental monitoring, earth-resource mapping, surveillance, and reconnaissance. The principle of SAR operation is that a radar antenna is attached to an aircraft or spacecraft. The antenna transmits electromagnetic pulses to the ground and records their echoes.

An output image is reconstructed from echoed data that is interpreted as a set of projections. The direct back-projection algorithm provides the most precise output image reconstruction and can compensate deviations in the flight track. A very high number of operations is required to reconstruct the output image because each pixel must analyze hundreds of these echoes. Therefore, graphic processing units, GPUs, are often used for this type of SAR data processing.

However, for applications with strict space and power requirements GPUs may not be an appropriate solution. For example, small unmanned aircraft systems may want to use the direct back-projection algorithm to compensate for deviations in the flight track but do not provide space and power for a computing system with a high performance GPU. Moreover, there are many applications that require the use of industrial and space grade components or demand a certain durability and reliability, which off-the-shelf GPUs may not be able to deliver. Hence, custom embedded systems need to be designed.

Although ASICs tend to provide the highest performance and lowest power consumption, the high setup and development costs make them viable for high

volume production only. This is a challenge for radar imaging systems that typically are both custom and low-volume. Alternatively, FPGAs have a much lower development cost than ASICs and are therefore increasingly being used in low and medium volume markets. The reconfigurability of FPGAs enables updates and modifications on systems that are already in operation. The logic integration of FPGAs has reached a point where a high number of processor cores, dedicated accelerators, and a large number of interfaces can be integrated on a single device. As many modern radar imaging systems successfully use FPGAs for signal processing already, it is convenient to integrate the data processing functionality in the same device.

In recent years several large multicore platforms have started to enter the market [6,61,111]. Although the architecture and the processing power of these platforms are suitable for radar data processing, a radar system will most probably still include an FPGA for dedicated signal processing tasks such as filtering and demodulation.

This case study describes how to map a specific SAR data processing application to a multicore system on an FPGA. It includes design of a scalable multicore system consisting of Tinuso processor cores and a 2D mesh interconnect. Data from the airborne POLARIS SAR is used for this case study [35]. This radar is currently used in the evaluation process of the European Space Agency's, *ESA*, BIOMASS candidate mission [40]. This mission aims for a P-band SAR satellite that provides global scale estimates for forest biomass.

The proposed system provides a number of advantages including system integration, power, scalability, customization, and the use of industrial and space grade components. As the power efficiency and logic capacity of FPGAs increases, they become an attractive technology for low-volume, large-scale systems. For example, Xilinx's Virtex-7 family comes with devices up to two million logic cells. These devices allow for combining the processing power of hundreds of processor cores on a single FPGA. Moreover, the same device can also host the digital front-end used for SAR signal processing. FPGAs provide flexible I/O that allows for connecting a multitude of data links and memory units to a single device.

We propose and advocate for a multicore system because it raises the abstraction level for the application programmer without facing the current performance drawbacks of high-level synthesis [90]. Moreover, mapping an application to multicore system significantly reduces development effort over a fully custom FPGA implementation [72]. The proposed system provides the ability for customizations at all levels. For example, it is possible to add processor cores, define special instructions, and change the interconnect link-width. FP-

GAs are available in industrial and space grade, which permits the use in rough environments and in space.

7.2 Synthetic Aperture Radar Application

Synthetic aperture radar is a form of imaging radar that is operated from a moving platform, typically an aircraft or satellite. SAR provides high quality radar mapping independent of light and weather conditions. Therefore, SAR is an attractive choice for a broad range of scientific and military applications such as environmental monitoring, earth-resource mapping, surveillance, and reconnaissance. The principle of SAR operation is that a radar antenna is attached to an aircraft or spacecraft and alternately transmits an electromagnetic pulse to the ground and receives echoes. As the radar moves along the track, it records its exact position, the energy level, and round trip delay of the received echoes. Signal and data processing is then applied to reconstruct an image of the scanned terrain.

The term synthetic aperture radar derives to the fact that a moving antenna effectively acts as a much larger antenna that can more accurately resolve distances in the flight direction. In radar technology, the term range refers to the slanted distance between the radar and a target or a scatterer that echoes the transmitted signal. Resolution in the range dimension of the image is typically accomplished by transmitting a pulse with a linearly frequency modulated pulse, called chirp pulse. A unique property of SAR is that resolution in flight direction, azimuth, is independent of range. It is therefore possible to obtain radar maps with a resolution of up to decimeters from very large distances.

Often frequency-domain algorithms are used for SAR image reconstruction [21]. These algorithms are based on the fast Fourier transform, *FFT*, technique and are computationally efficient to implement. A limitation of these algorithms is the approximations involved and the high sensitivity to non-linear deviations in the flight track. This is a major problem for low frequency SAR where a long synthetic aperture is required to obtain high output image quality [113]. Direct back-projection is a time-domain algorithm that can adapt to a general geometry of the synthetic aperture and therefore compensate for deviations in the flight track. However, the high image quality of the direct back-projection algorithm comes at the cost of high computation power.

The received echo of each transmitted radar pulse is stored in a one-dimensional array, called range line or projection. It contains values that represent the reflected energy and propagation delay of all points on the ground swath illumi-

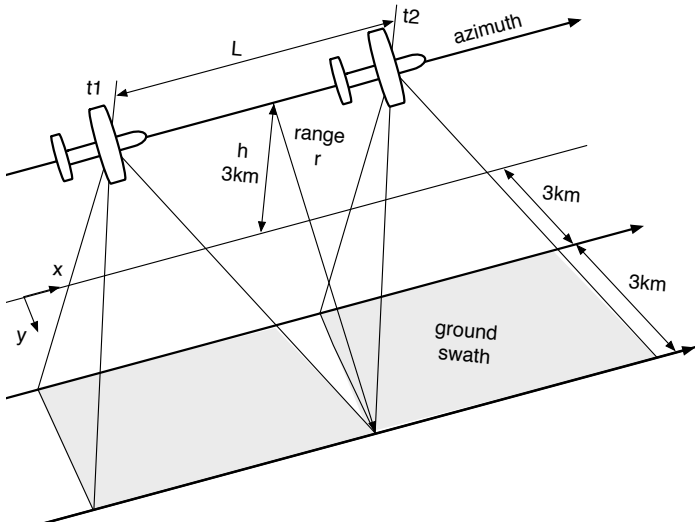


Figure 7.1: SAR system overview

nated by this pulse. To reconstruct a projection image, the computation of each projected pixel must consider all range lines that possibly contain an echo from the corresponding point on the ground. The energy contribution of each range line for each pixel is computed and coherently accumulated. As the resolution of a SAR image depends on the length of the synthetic aperture, the upper limit of which is the along track length of the footprint illuminated by the antenna, hundreds up to thousands of range lines contribute to a single output pixel. Thus, the number of operations required for reconstructing an output image with NxN pixels and N range lines is proportional to N^3.

7.2.1 Case Study Application

This case study considers SAR data processing for the POLARIS system, which is an ice sounding radar developed at DTU [35]. The radar operates at 435 MHz with a chirp pulse bandwidth of 85 MHz, which allows for a range resolution of 2 meters. The radar is mounted on an airplane that flies at an altitude of 3000m. Real-time data processing must be provided for a 3000m wide ground swath as shown in Figure 7.1. The relatively low radar frequency and the small antenna cause a long synthetic aperture of 700 up to 1100 meters depending on

```
for each pixel do
    for each range line do
        calculate round-trip delay and fetch data samples from memory
        reconstruct echo signal and interpolate the energy contribution do
        amplitude weighting and phase correction accumulate energy
        contributions of each range line
    end
end
```

Algorithm 1: Pseudo code for SAR data processing with direct back-projection

the slant range. The synthetic aperture, L, is shown in Figure 7.1.

This study assumes an oversampling ratio of 1.25 and considers 50 range lines per second to avoid aliasing. The length of the synthetic aperture for objects that are closer to the radar is smaller than for objects further away. Hence, for each pixel it is necessary to calculate the energy components of 437 up to 688 range lines. 1500 Pixels in range dimension are required to map a 3000m wide ground swath with a resolution of 2 meters. Given a flight speed of 80 m/s, it is necessary to compute 60'000 pixels per second to provide real-time processing. For the given application, the long synthetic aperture length of up to 1100 meters and the relatively low flight speed of 80 m/s require recorded data of up to 13.75 seconds flight time to reconstruct a single pixel. Hence, it is evident to use a image reconstruction algorithm that can compensate for deviations in the flight track. Figure 7.2 shows a SAR image of the Kangerlussuag airport area, taken by POLARIS P-band SAR in June 2012.

7.2.2 Direct Back-Projection

Direct back-projection is a well-known technique for focusing wide-band SAR data. All energy components of a point in the scene are coherently accumulated to reconstruct a projected pixel. The algorithm is described by Equation 7.1 [68, 113].

$$s_0(x_T, y_T) = \sum_{|x_R - x_T| < \frac{L}{2}} s_i[x_R, \tau_{rt}] exp(j2\pi f_c \tau_{rt}) \tag{7.1}$$

$S_0(x_T, y_T)$ is a focused output pixel at a given position. (x_R, y_R) is the position of the radar sensor. $s_i[x, \tau]$ represents the input data ordered by the

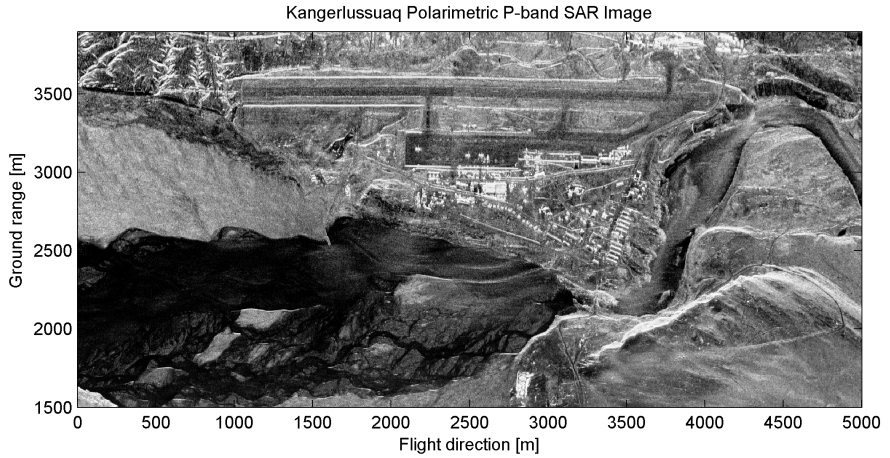


Figure 7.2: POLARIS P-band SAR image of the Kangerlussuag airport

position in azimuth and time delay. L is the length of the synthetic aperture, f_c is the radars center frequency and τ_{rt} defines the round trip delay of the transmitted signal. Thus, for each output pixel a summation over all input range lines within the synthetic aperture is required.

7.2.3 POLARIS Data Processing

SAR data processing for POLARIS is described by Algorithm 1. It includes direct back-projection, interpolation, amplitude weighting, and phase correction.

The input of the algorithm includes a number of pixel coordinates that need to be calculated, plus additional flight track information. In a first step the round trip delay of the transmitted signal is calculated. This corresponds to the signals traveling time from the antenna to a point on the ground swath and back to the receiver. The round trip delay specifies a sample in the range line.

An interpolation step is needed because the round trip delay mostly points to a value that lies between two samples. Interpolation distributes the energy components in a range line to the respective pixels. Hence, it is necessary to reconstruct the received echo signal as accurately as possible to obtain a high output image quality.

Interpolation kernels based on the *sinc* function show a very low interpolation error and are well suited for image projection applications [74]. The

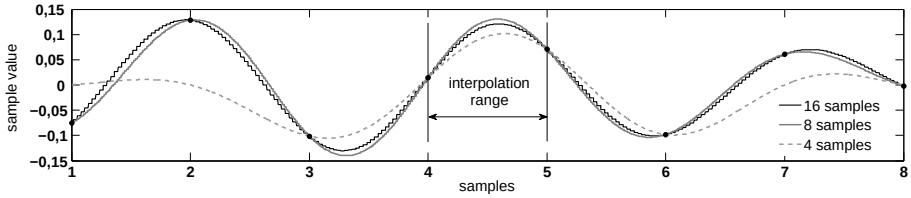


Figure 7.3: Sinc interpolation

sinc function is defined as $sin(x)/x$. We use the *sinc* interpolation method to reconstruct the sampled analogue waveform. This allows for interpolating intermediate points between data samples. Equation 7.2 provides the reconstruction of an equally spaced discrete signal with *sinc* functions.

$$x(t) = \sum_{i=-\infty}^{\infty} xn\ sinc(\frac{\pi}{T}(t - n\tau)) \tag{7.2}$$

In practice the sinc function is truncated. The more samples we consider, the more accurate the reconstructed waveform gets. For this case study the *sinc* function is implemented with a lookup-table with 1024 single precision floating point entries as it fits well into a block RAM. Figure 7.3 shows *sinc* interpolated waveforms with 4, 8, and 16 samples. The interpolated waveform becomes more accurate the more samples are used. Contrary, as we increase the range of the *sinc* function, its accuracy is lowered due to the limited number of entries in the lookup-table. This trade-off can be seen on the rough waveform of the interpolation with 16 samples. An interpolation with eight samples is used as it provides sufficient accuracy and a lookup-table with 1024 entries is sufficient.

The next step of the algorithm applies phase correction and amplitude weighting. For this computation, it is necessary to extract the phase and amplitude information of the interpolated IQ signal. This is done using *sine* and *cosine* functions. A Hamming window function is used for weighting the amplitude. Finally, the value of the pixel is determined by coherently accumulating the energy components of all range lines, within the synthetic aperture.

This SAR data processing application provides a high level of parallelism for reconstructing the output image. To optimize application performance, we exploit parallelism at task-level. We define a task as the computation of a single output pixel. According to Algorithm 1, the computation of each range line

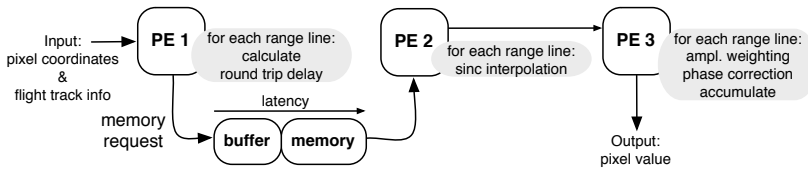


Figure 7.4: Software pipelined SAR data processing application

includes a memory request. As there is a memory operation in the time-critical path of the algorithm, the performance greatly depends on memory latency.

We evaluate two implementations of this application. The first implementation assumes a low memory latency and simultaneously executes Algorithm 1 on a number of parallel processing elements. This implementation is called homogeneous SAR. The second implementation attempts to hide memory latency to achieve a higher performance.

GPUs employ hardware multi-threading to successfully hide memory latency. Hardware multi-threading requires registers and memory to maintain the contexts of all threads. Due to the limited hardware resources in FPGAs software pipelining is applied to tolerate memory latency. The task of computing an output pixel is split up in three sub-tasks that run on individual processing elements as shown in Figure 7.4. The first sub-task calculates the round trip delay for each range line and thereby determines which memory blocks need to be fetched. These memory blocks are then forwarded to the processor that runs the second sub-task, which includes interpolation. The interpolated values are then sent to the core that runs the third sub-task. The advantage of this technique is twofold: it hides memory latency and simplifies the processing elements. However, the execution time of the sub-task needs to be balanced. The sub-task with the longest execution time defines the throughput of the whole system.

In this software pipelined application there is direct communication between processing elements. To efficiently run this application, a target platform must provide a message passing communication scheme.

7.3 Related Work

EMISAR is an airborne SAR developed at the Danish Technical University, DTU, which provides a resolution of 2x2m [26]. It was mainly used for research

in remote-sensing techniques and to collect high-quality SAR data. EMISAR operates in the L- and C-band, at 1.25 GHz and 5.3 GHz. A high frequency and a relatively large antenna lead to a quite short aperture length, which limits the number of echoes that need to be considered to compute a pixel of the output image. In the sensor sub system, a real-time processor is able to produce an output image of a single channel [34]. This was mainly used to check the data acquisition. An offline system then did the high-quality processing of the data that was stored on HDDT tapes. Both the real-time processor and the offline processor are based on the range-Doppler algorithm [117], as the relative bandwidth is so small that the back-projection algorithm is not required.

POLARIS is an ice sounding radar developed at DTU [35]. It was initially built to assess the potential and feasibility of a space-based P-band ice sounding mission. It operates in the P-band at 435MHz. A radar with a low frequency is used to avoid excessive attenuation of transmitted and reflected signals in the ice. The on-board signal processing supports real-time visualization at a coarse resolution only. This is sufficient to calibrate the system. Final data processing is done offline. POLARIS is currently used as a SAR system to support the evaluation of the ESA's BIOMASS candidate Earth Explorer mission. The long wavelength of P-band SAR has a higher saturation threshold to biomass than radars operating in a higher frequency band. This ESA mission aims for a P-band SAR satellite that provides global scale estimates for forest biomass.

SAR data processing always has been a challenge due to the huge input data and large amount of processing per pixel. In the past, dedicated hardware and large computing clusters were used, e.g. the EMISAR real-time processor is a pipeline processor including about 20 programmable signal processing elements, each with 8 digital signal processors, *DSPs* [34]. Modern systems, however, make use of accelerators of various forms.

The highly parallel architecture of GPUs is well suited to accelerate SAR data processing. Fasih and Hartley present a SAR back-projection implementation in CUDA [43]. The input data array of SAR systems is typically too large to fit in graphics memory. Their implementation allows for partitioning the input data matrix in small data blocks that are used to calculate a block of neighboring pixels. The advantage of this approach is that data blocks are small enough to be placed in the GPU's texture memory, which enables a low access latency. This texture unit also provides hardware-accelerated linear interpolation. Moreover, the algorithm makes use of spatial locality by reusing input data to compute neighboring pixels. They use two public spotlight SAR data-sets to evaluate their implementation [102], [22]. The GPU accelerated im-

plementation performs up to 344 times better than on a single threaded desktop CPU.

GPU accelerated data-processing is also done for lower frequency SARs. The property of radars with a low frequency is the ability to better characterize the surface of scanned objects. However, the data that need to be processed is larger due to the high number of echoes that need to be considered. Some application produces data arrays that do not fit in the texture memory. Hence, their GPU implementation only performs up to 4 times better than on a desktop CPU [15].

FPGA accelerated SAR data processing has been proposed previously [30]. They use a host machine and an FPGA accelerator connected over a PCI interface. The host machine provides the FPGA with preprocessed data and application specific information at run-time. The back-projection algorithm is then implemented on the FPGA. They reach a speedup of 200x over a pure software implementation. However, their system is completely different from our proposed approach. While our multicore system consists of generic processing cores and interconnect, they use dedicated hardware blocks only. We provide interpolation to get a better estimate of the energy contribution of each echo, which is not done in their system. They use fixed-point arithmetic while we do all data processing in floating-point arithmetic. Finally, we implement amplitude weighting and phase correction, which is not supported in their design either.

7.4 System Architecture

We aim for a system as shown in Figure 7.5 to integrate SAR signal and data processing. Signal processing is done partially in an analogue front-end where the received echo is mixed down to base-band, IQ de-modulated, and A/D converted. A/D converted antenna signals are fed into the FPGA. The digital front-end in the FPGA may be used for digital filtering and data preprocessing. Preprocessed data then may be stored in off-chip memory or directly forwarded to processing elements. The image reconstruction application is mapped to a number of parallel processing elements that communicate over an interconnect with a memory controller. Given the high number of I/Os in modern FPGA's several A/D converters and off-chip memories can be connected to a single device.

Each processing element must provide a broad range of integer and floating-point operations. Therefore, synthesizable processor cores are well suited to run

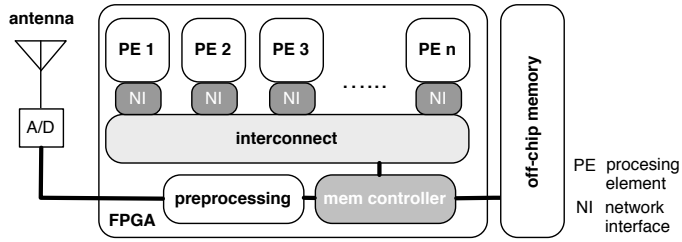


Figure 7.5: Block diagram of a SAR signal and data processing system

this algorithm. For the software pipelined application it would be possible to use dedicated hardware blocks. However, we prefer a processor based solution because it raises the abstraction level for the application programmer.

7.4.1 Processing Element

We decided to use instances of the Tinuso processor architecture as processing elements. Tinuso is a three operand, load-store architecture with a fixed instruction word length of 32-bits. It comes with a super-pipelined, single issue, in-order pipeline that is optimized for a high instruction throughput when implemented on an FPGA. The pipeline is fully exposed to software where all types of hazards need to be considered. The architecture supports predicated execution to reduce the branch penalty. Tinuso is a lightweight architecture with a small instruction set that can easily be extended. Given the high instruction throughput, the small hardware footprint, and the ability to extend the design, Tinuso is an attractive choice for our multicore system. However, a number of application specific modifications and extensions to the initial design are required to efficiently run SAR back-projection algorithms. Figure 7.6 shows the modified eight-stage Tinuso pipeline. The design includes function lookup-tables and a single precision floating-point unit.

Function Lookup-Tables

SAR data processing as described in Section 7.2.3 requires specific mathematical functions for interpolation and amplitude weighting.

Implementing a mathematical function in an FPGA is a trade-off between accuracy, clock speed and utilized resources. For example, the coordinate rotation digital computer algorithm, *CORDIC* is often used to implement trigonometric

blocks in hardware [99]. The Xilinx core generator and various other high level VHDL code generation tools can be used to easily configure and implement arithmetic blocks. Depending on the desired system clock frequency and the available hardware resources, one or more pipeline stages are used. A dedicated *sinc* function block is very costly as it includes *sine* function and division. Thus, we decided to use function lookup-tables.

Modern FPGAs come with extensive memory resources in the form of synchronous static SRAM blocks, *block RAMs*. Lookup-tables implemented in block RAMs are an attractive design choice because they do not occupy configurable logic blocks. However, function lookup-tables may suffer form accuracy and are therefore rarely used in general-purpose high performance processors. We build application specific processor extensions and know the application exactly. Hence, we are able to limit the range of the functions that are placed in the lookup-tables and thereby attain a sufficient accuracy.

Current state-of-the-art FPGAs provide pipelined memory resources for fast designs. Tinuso uses such pipelined block RAM configurations already to implement register file and caches. The block RAM size in Xilinx Virtex-6 FPGAs is 36 kilobits, which allows for a lookup-table with 1024 data words. We implemented three function lookup-tables for *sinc*, *sine*, and a Hamming window function. The integration in the pipeline is shown in Figure 7.6. The lookup-tables are placed in the execution stage, parallel to the ALU. Forwarding in the execution stages is permitted from the ALU to the function lookup-tables only.

Floating-Point Unit, FPU

In this case study, the echoed signal is demodulated in IQ format. Input data consists of a pair of single precision floating-point numbers. All computation is done using single precision floating-point arithmetic. FPU functions can be emulated by a sequence of integer operations, which saves the added hardware cost of an FPU. As this computation is significantly slower, there is a higher number of parallel processing elements required to yield the same performance. However, the hardware cost of an FPU is smaller than the costs for additional processing elements. Thus, the Tinuso pipeline is extended with a floating-point unit. The Xilinx floating-point core is used to implement an FPU with the following operations: addition, subtraction, multiplication, division, and square root [119]. Tinuso uses lookup-tables to implement arithmetic functions where an integer index is used to look up floating-point values. Hence, dedicated function blocks that allow for efficiently converting floating-point numbers to fixed-point numbers and vice versa are included in the design.

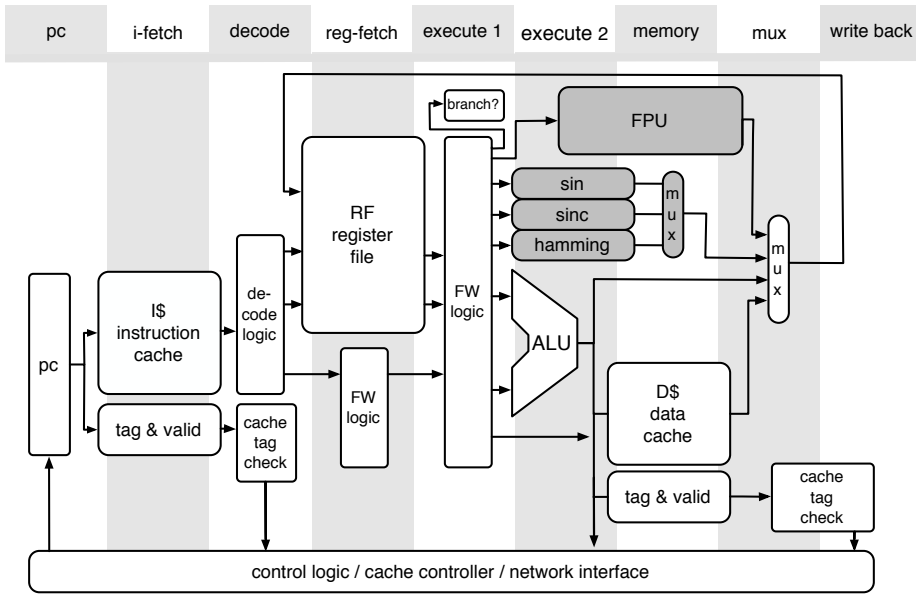


Figure 7.6: Pipeline sketch of a Tinuso implementation with FPU and function lookup-tables

Floating-point blocks built with the Xilinx floating-point operator can be customized for operation, word length, latency, and interface. Table 7.1 shows the design parameters for each of the implemented operations. The hardware costs are listed in terms of CLB lookup-tables, *LUTs* and DSP-slices. Latency specifies the number of clock cycles to perform a given operation. The latency corresponds to the number of pipeline stages used to implement the operation. Increasing the number of pipeline stages leads to a faster design at the cost of a larger hardware footprint. For all FPU operations, the number of pipeline-stages is adjusted to attain a system clock frequency of 330 MHz on a Xilinx Virtex-7 device. The term cycles per operation describes the minimum number of cycles that must elapse between instructions of the same operation. If the number of cycles per operations is increased, the hardware footprint is lowered.

Some operations, such as division and square root, are used infrequently. Hence, no performance is lost when these operations are optimized for a low area footprint. Forwarding is not supported for any floating-point instruction.

Table 7.1: Floating-point operation configuration

Function	LUTs	DSP	latency	cycles
floating-point add / sub	266	2	8	1
floating-point multiply	178	2	6	1
floating-point division	313	0	28	26
floating-point sqrt	174	0	28	25
float-to-int	178	0	6	1
int-to-float	201	0	6	1

Operations such as addition, subtraction, and multiplication are configured to use DSP slices to minimize the CLB utilization. A FPU based on the Xilinx Floating-Point core complies with most of the IEEE-754 Standard [88]. In particular, Tinuso's FPU does not support de-normalized numbers and always uses the round to nearest rounding mode.

Memory Architecture and Network Interface

For this case study, the network interface is modified for the given application. To keep the hardware footprint low, it only implements functionality strictly required by the application. For example, the implementation of the back-projection algorithm transfers data explicitly. Hence, the data caching mechanism is not used. Instead, message passing communication is required to support efficient communication between processing elements. The network interface contains a small FIFO buffer for outgoing messages. The processor core is writing to that FIFO and triggers a message transfer. The data cache is modified that the state machine in the network interface places the incoming data packets directly in the data cache. Tag and tag check logic of the data cache are removed, which also simplifies the cache controller state machine. An identifier in the packet header specifies where data is placed. Once the complete incoming package is written to the data cache, a status register is set. The processor core is polling this status register to check whether a packet has arrived or not.

7.4.2 Interconnection Network

The interconnection network plays a vital role in the performance of applications with a high communication to computation ratio. The SAR data pro-

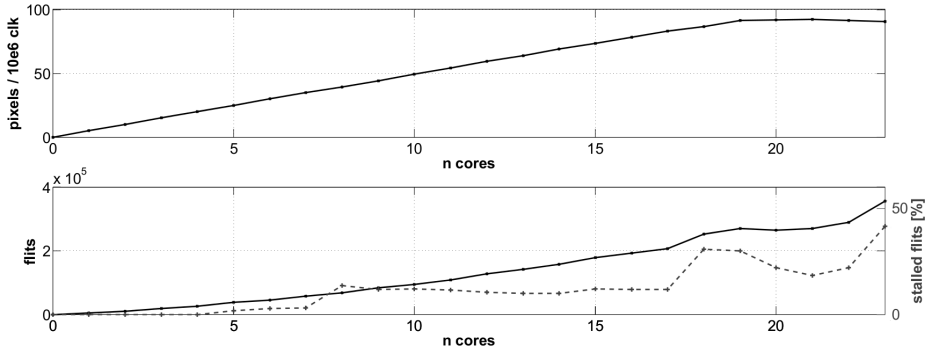


Figure 7.7: Performance scaling and network traffic of the SAR application

cessing application requires a high throughput, low latency, and deadlock free network-on-chip. We use the wormhole-switched router with five bidirectional ports for a 2D mesh interconnect as described in Chapter 5. As processor cores and interconnect operate in the same clock domain, we decided for the pipelined router implementation to obtain a highest possible system clock frequency. Profiling the network traffic showed that the regular traffic pattern of the SAR data processing application does not benefit from the round-robin arbitration logic. We therefore have disabled this feature to keep the hardware footprint low.

Given the regular traffic patterns of this case study application it would have been possible to implement an application optimized network topology rather than a regular 2D mesh. As most traffic goes to memory, a star network topology with a central memory would probably have required fewer hardware resources to implement. However, a star network topology with a high number of nodes does not fit well on the FPGA fabric and therefore would require a high number of buffers to obtain a high system clock frequency. We decided for a regular 2D mesh because it maps well to the FPGA fabric and because it scales better.

7.5 Hardware Organization

We have now introduced processing elements, interconnect, and described how to map the application to hardware. At this point, we do not yet know how many parallel processing elements to employ in the final system. We therefore measure the scalability of the SAR data processing algorithm. In a massively

parallel system typically memory access bandwidth is the limiting factor.

We measure the scalability by running a number of parallel instances of the algorithm and record the total performance of the system. We use a network with 25 nodes whereas one node is used for the memory controller. We run a set of experiments where we populate 1 up to 24 nodes with processor cores. All memory requests go to a memory controller that is connected to a corner node of the interconnect. As our implementation is sensitive to memory latency, we simulate a memory controller using fast synchronous SRAM.

We argue for such a design as in recent years quad data rate SRAMS, *QDR*, started to penetrate the market. In QDR, data transfers use both rising and falling clock edges. For example, data can be read on the rising clock edge while writing is done on the falling clock edge. This allows for simultaneous read and write burst accesses. This is an important feature for SAR data processing because it allows for storing incoming data at run-time. Xilinx provides an application note, where implementation and timing details of a QDR SRAM interface are described [33].

The upper part of Figure 7.7 shows the performance of the system for a variable number of parallel instances. For this test setup, we define performance as the number of pixels calculated per 1 million clock cycles, whereas each pixel includes data of 500 range lines. We observe an almost linear performance increase until a system size of 19 processor cores. We measure an evenly distributed workload among the processing elements.

A static arbitration scheme may lead to a lower network throughput and some processor cores getting isolated [87]. We observe this behavior when the application runs on 20 or more parallel processors. Our static arbitration scheme prioritizes packets that are in the network over packets that the home node tries to inject. With increasing network traffic it becomes more difficult for processor cores to inject packets into the network. Thus, some cores get completely isolated and run idle as they cannot inject their memory requests.

The lower part of Figure 7.7 shows the number of flits in the network and the percentage of flits that are blocked due to backpressure signaling. We observe a significant increase in blocked flits when 18 cores run the application in parallel. Since real-time data processing is required we do not want to push the system to the limit and decided for 16 processing cores per memory controller. A setup consisting of 16 processing cores and one memory controller is considered as a sub-system.

One of the advantages of FPGAs is the flexible I/O capabilities that allows for connecting a multitude of memory blocks to a single device. We compose a multicore consisting of four sub-systems to provide sufficient computing power

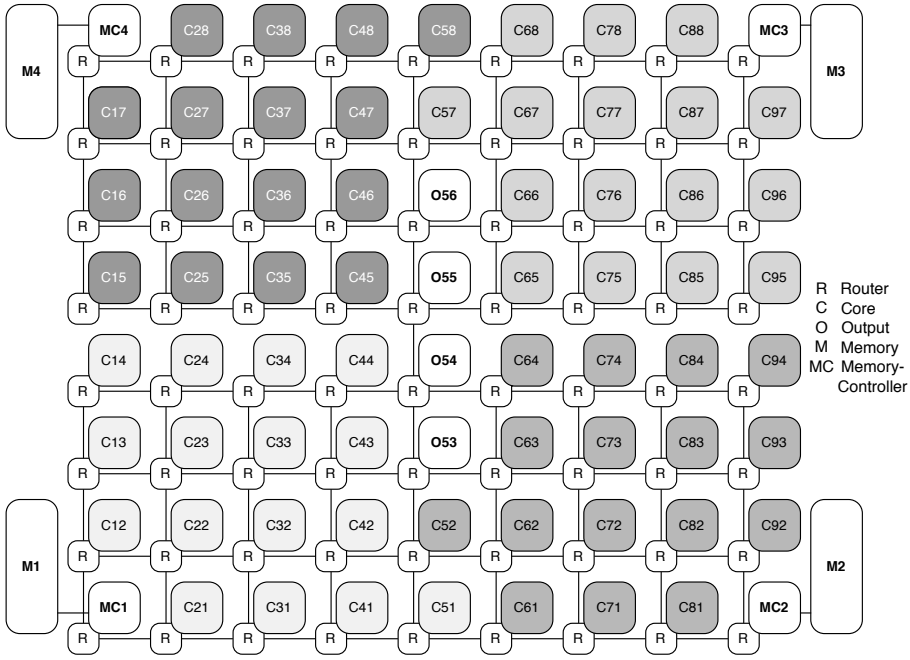


Figure 7.8: Overview of the complete multicore system

for the case study application. The final system comes with 64 processor cores and 4 memory controllers. We use a NoC with 72 nodes as shown in Figure 7.8. Processing elements are depicted in different gray tones to mark the four subsystems. In this experimental setup there are four nodes in the center of the network that are not included in the image reconstruction. In the final system, these processors can be used to implement control and monitoring functionality.

7.6 Results

We evaluate the proposed multi-core system by running the SAR data processing application. First, we derive the system's maximum clock frequency and the required hardware resources for various FPGA families. Second, we evaluate

Table 7.2: Overview of hardware resource usage and clock frequency

FPGA Family	Grade	F.Max	Device	Utilization
		[MHz]		[%]
Xilinx Virtex-7	-3	300	xc7vx550t	60
Xilinx Virtex-7	-2	260	xc7vx550t	60
Xilinx Virtex-6	-2	250	xc6vhx550t	60

the processing performance, network traffic, and application mapping.

7.6.1 Speed and Resources

Table 7.2 shows the speed and resource results of a multi-core system as shown in Figure 7.8. Memory controllers were simulated only, therefore they are not included in these results. Each of the 64 processing cores is equipped with a FPU that supports all operations listed in table 7.1. We utilize about 60% of an FPGA with 550'000 logic cells. We measure a maximal system clock frequency of 300 MHz on Virtex-7 device of the fastest speed grade. The speed and resource results are based on Xilinx ISE 13.4 "place and route" report. We use Xilinx SmartXplorer to run multiple implementation flows using varying sets of implementation properties and strategies until timing closure is achieved.

7.6.2 Performance and Network Traffic

In this subsection, we evaluate the performance of the homogeneous SAR application. As the complete hardware is implemented in VHDL we use the open-source simulator GHDL for our functional experiments. All 64 parallel processor cores in the system implement Algorithm 1 to compute pixels of the output image.

The Tinuso architecture comes with a total of 128 registers. To speed up the application, we use many of these registers to store pre-computed constants, packet headers, and intermediate results. We used assembly language programming to optimize instruction scheduling. This allows for computing the energy content of a range line in 350 clock cycles. The memory controllers are integrated in a test bench that simulates synchronous SRAM memory. Memory controllers are connected to regular network nodes. As they receive a memory request message, memory address and destination node is decoded. Then, the

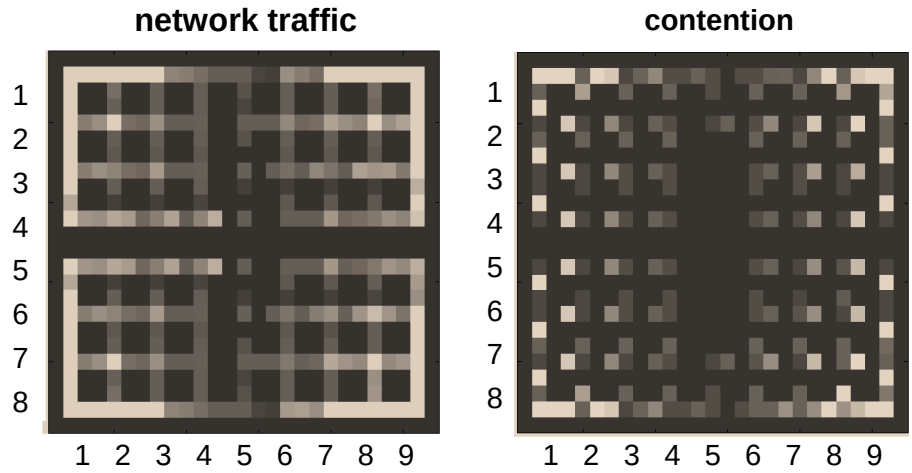


Figure 7.9: Network traffic and stalled flits of a homogeneous SAR application

desired data is fetched from memory and then sent to the destination node. We assign a fixed latency of 5 clock cycles to this memory model.

Network traffic includes memory requests and the input and output messages of the algorithm. The left part of Figure 7.9 illustrates the network traffic of a system configuration as described in Figure 7.8. We obtain this data by counting the number of flits in the network while we run the application. We extend the router ports with counters to record the network traffic. When the GHDL simulation has completed these counter values are stored in a file. We then use MATLAB to plot the data of the hardware simulation. Network traffic of the initial instruction fetching sequence is not included in this data.

Each network node contributes with an array of 3x3 pixels to the graph. Routers are represented in this array with five pixels that form a cross shape. The four outermost pixels correspond to the number of flits passing through the N,E,S,W ports. The center point pixel represents all flits that pass through the router. The right part of Figure 7.9 illustrates the number of clock cycles where backpressure is applied. Hence, it marks stalled flits in the network. Light colors in graph indicate high network traffic respectively a high number of stalled flits.

We observe the highest traffic in the corners of the system where the memory controllers are located. It is visible that an XY-routing scheme is used given the high traffic at the outer edges of the system. We see stalled flits in almost all

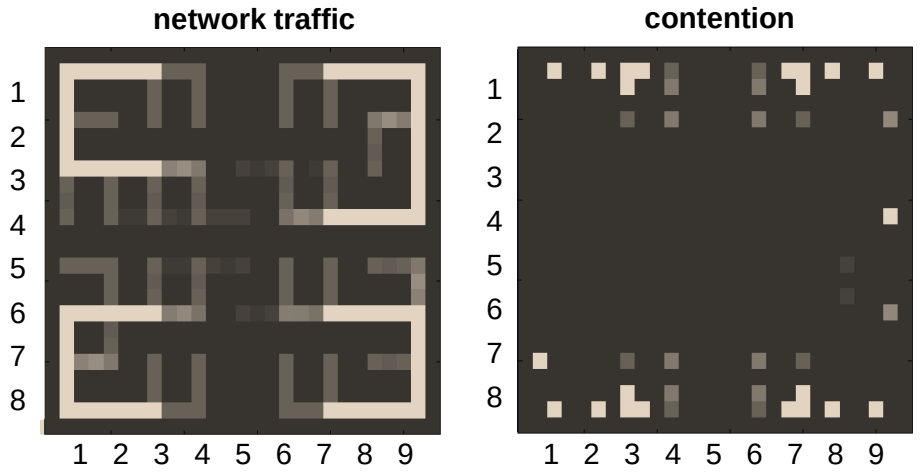


Figure 7.10: Network traffic and stalled flits of a software pipelined SAR application

network nodes in the system whereas the frequency of occurrence is increasing towards the memory cores. In the simulation, all processor cores start the application simultaneously and therefore inject their memory requests in the network at the same time. Memory controllers allow only one request at the time, therefore parallel memory requests have to execute sequentially. This leads to stalled flits all over the network, as illustrated in Figure 7.9. Once the initial memory requests are serialized, the applications in the individual processing elements execute time-shifted and fewer simultaneous parallel memory requests occur.

7.6.3 Software Pipelined SAR

An application mapping as described in Section 7.6.2 is prone to memory latency as processor cores remain idle until memory requests are resolved. This is a limitation when memory with higher latencies is used. We therefore proposed a software pipelined implantation as shown in Figure 7.4. We split the tasks of computing an output pixel up in three sub-tasks that execute on three different processing elements. We run five parallel instances of the software pipelined application on each sub-system. Hence, one processing core per sub-system is

not used.

In the software pipelined approach, memory latency can be hidden because the processor elements do not wait until memory requests are executed. Moreover, this approach allows for simplifying the individual processing elements. For example, the floating-point square root operation is only used in the processor core that computes the round trip delay. Thus, this operation is omitted in the processing elements that run the other sub-tasks. The hardware utilization of the homogeneous and the pipelined approach are listed in table 7.4. The software pipelined approach leads to significant hardware savings of 20%. However, the software pipelined implementation includes 15 processing elements per sub system only. If we also populate the 16th core, we achieve a reduction of the hardware footprint by 14%.

The network traffic of the pipelined application is illustrated in the left part of Figure 7.10. We observe a significant difference to the homogeneous implementation because the network traffic is heavily dependent on the application mapping. The sub-figure at the right side shows the stalled flits in the network. In the pipelined SAR application, there are only 5 processor cores per sub-system that send requests to the memory controller. Hence packet collisions occur at fewer places.

Table 7.3 lists the performance of two homogeneous SARs and the software pipelined SAR implementation. For this experiment, we define performance as the number of pixels calculated per million clock cycles, whereas each pixel includes data of 500 range lines. We compare homogeneous SAR applications that use 4 and 8 samples for *sinc* interpolation. The reduction from 8 to 4 samples lowers the network traffic by 20% but only leads to performance increase of less than 1%. This can be explained, as the number of memory request is the same for both implementations. Requesting fewer samples leads to shorter reply messages sent by the memory controller. However, the communication overhead remains the same and no significant speedup is achieved. The software pipelined SAR application provides a slightly lower performance than the other implementations because it employs 15 processing cores per sub-system only.

For the pipelined application, we measure a significantly higher number of flits that are stalled due to backpressure flow control. In the homogeneous implementation, the initial serialization of parallel memory requests leads to a time-shifted execution, which results in a low number of stalled flits for the successive memory requests. The five instances of the application execute simultaneously at any time because the processor cores do not wait for memory requests to be resolved. Hence, the serialization of simultaneous parallel memory request needs to be done again and again, which causes a high number of

Table 7.3: Overview of performance and network traffic of various SAR implementations

Implementation	Performance	Traffic	Backpressure
	[px / 10e6 clk]	[flits / clk]	[%]
Homogeneous SAR 8-sample	310	16,40	19,75
Homogeneous SAR 4-sample	312	13,12	13,45
Pipelined SAR 8-sample	290	16,99	27,07

Table 7.4: Overview of hardware resource usage of the homogeneous and pipelined SAR implementations

Hardware resource	Homogeneous SAR	Pipelined SAR	Diff
Number of Slice Registers	200k	190k	-5%
Number of Slice LUTs	209k	167k	-20%
Number of RAMB18E1	896	600	-33%
Number of DSP48E1s	256	240	-6%

stalled flits.

According to Section 7.2.1, we provide real-time data processing for the case study application when we are able to calculate 60'000 pixels per second. Each pixel has to consider 437 up to 688 range lines, depending on the length of the synthetic aperture. We measure an execution time of 905 ms to compute 60'000 pixels using the pipelined SAR application on a Virtex-7 device of a medium speed grade. The Xilinx Pocket Power Estimator computes a power dissipation of about 10 watts for the complete multicore fabric.

7.7 Conclusions

This chapter described how to map a SAR data processing application on a multicore on an FPGA. We implemented a multicore system consisting of 64 Tinuso processor cores and a 2D mesh based network-on-chip. The design is optimized for a high data throughput and low network latency and utilizes 60% of the hardware resources of a Xilinx Virtex-7 device with 550 thousand logic cells and consumes about 10 watt.

We evaluate the system by simulating data processing for the airborne POLARIS P-Band SAR. The computationally heavy direct back-projection algo-

rithm is used to reconstruct a high quality output image. The proposed system provides real-time data processing for a 3000m wide swath with a resolution of 2x2 meters. Software pipelining is a method to achieve certain insensitivity to memory latency and reduces the hardware footprint by 14%.

We show that real-time data processing for the POLARIS SAR can be done on a multicore system on an FPGA. FPGAs are often used to implement the digital front-end of a radar system. Hence, it is possible to combine SAR signal and data processing in a single device. The use of synthesizable processor cores raises the abstraction level for the application programmer. This is of particular interest when the application needs to adapt quickly to flight and scene properties. We conclude that multicore systems on FPGA are an attractive choice for application with strictly constrained space and power budgets.

The device cost of large scale high performance FPGAs is very high. Depending on the application, large embedded multicore platforms from Tilera, Kalray or Adapteva might be viable alternatives as they provide a highly parallel architectures with a large processing power at a low power budget. However, the market of embedded multicore architectures is very dynamic and market shares might change quickly when one of the major hardware producers comes up with a commercial embedded multicore architecture. Hence, it is not clear which architectures will still be on the market in a few years. This is a problem for systems that require a long product lifetime. The advantage of synthesizable multicore systems is that they can easily be ported to new FPGA architectures to ensure a very long lifetime of a product.

Chapter 8

Programming Model and Runtime System

In the previous chapters Tinuso multicore systems were introduced. It was described how to optimize the processor pipeline and network for high performance on FPGA and how to map an imaging radar application on a system with 64 processor cores. However, the programmer needs to be aware of many architectural details to successfully exploit the parallel architecture, which is a difficult task. We therefore need a programming language, a programming model, and a runtime system to provide the programmer with a suitable abstraction of the underlying computer system, which allows for easily expressing parallelism and an efficient execution of parallel programs.

Section 8.1 introduces and motivates programming language, programming model, and runtime system for Tinuso multicore systems. Section 8.2 discusses related work on parallel programming models and runtime systems. Section 8.3 presents basic semantics for creating tasks and scheduling. Section 8.4 describes the envisioned implementation of the runtime system and considers memory coherency. Section 8.5 provides example programs that describe how to map and optimize a parallel matrix multiplication and the direct back-projection algorithm to Tinuso multicore systems. Section 8.6 considers costs of the proposed runtime system and the coherency mechanism. Section 8.7 concludes this chapter with some final remarks.

8.1 Introduction

Tinuso is a scalable multicore architecture that is optimized for high performance when implemented on FPGA. For example, Xilinx's Virtex 7 family comes with devices up to two million logic cells that allow for multicore systems with up to 480 Tinuso processor cores [118]. When the entire multicore system operates at a clock frequency of 300 MHz and all processor cores execute one instruction each clock cycle, we obtain a total performance of 144 Giga integer operations executed in one second, *GIPS*.

The cache controller implicitly generates and processes memory requests and a lightweight communication interface allows for explicit communication with other processing elements. However, it remains a challenge to program these systems mainly because of two reasons. First, all processing elements share a global shared memory address space, but there is no hardware mechanism implemented that keeps memory consistent. Data written by one processor core will not automatically be visible to the other cores. Thus, the programmer has to manually ensure that the program operates on the correct copy of a data, which is a daunting task. Second, Tinuso's communication hardware primitives are designed to enable a high performance communication management. However, using these low-level send and receive primitives results in a complicated programming process [47].

We therefore need a suitable programming language and programming model to support the programmer expressing and exploiting parallelism. We research opportunities on how to implement a runtime system and hardware primitives that allow for an efficient execution of parallel programs. Moreover, a mechanism must be developed, which maintains memory coherency while not limiting the scalability of the system.

A programming language for Tinuso multicore systems has to support the programmers in expressing parallelism. Tinuso multicore systems may be heterogeneous and certain processing elements might be more suitable to execute a given part of an application than others. For example, as double precision floating point operations are costly to implement in hardware, only a few processor cores in a multicore system may be equipped with this functionality. Hence, a programming language has to allow for binding program code with frequent double precision floating point operations to dedicated processing cores.

A plethora of parallel programming languages exist and we do not want to introduce another one. Preferably, we want to use the existing GCC based compiler infrastructure for Tinuso. A widely adopted approach is to extend standard programming languages with specific keywords to express parallelism.

We therefore use C programming languages and adopt some OpenMP directives to express parallelism. However, at this point we do not aim for a complete implementation of OpenMP nor to be fully compliant with the OpenMP standard. Instead, we focus on research in programming models and memory hierarchy for embedded multicore systems.

The programming model must be able to exploit the parallelism of the underlying platform and support the programmer to easily express parallelism to enable a high productivity. A commonly used abstraction level is to consider parallel applications as a set of tasks. On a single processor system, tasks are executed sequentially. As there are more processor cores available, the programming model aims to exploit the parallel architecture by executing tasks concurrently. Thus, there will be processor cores that create tasks, which then may be executed on a different processor core. The task scheduler distributes parallel tasks to processor cores or puts them in a queue and let idle processor cores steal these tasks, this is referred as work-stealing. Work-stealing is applied to achieve an optimal load balance in a multicore system.

The runtime system implements functionality of programming language and programming model. As we have access to all of Tinuso's VHDL sources, it is possible to improve performance by implementing commonly used and performance critical functionality in hardware while corner cases are handled in software. For example, hardware primitives may allow for efficient communication or manage memory resources.

In multicore systems, where processor cores maintain caches of a shared memory resource there may arise problems with inconsistent data. In Tinuso, there is no restriction on memory addresses in the hardware. The global address space is shared among all processing elements of which each can access the entire address space. The current implementation of Tinuso comes with both instruction and data caches. Once a processor core writes a memory location it is only updated in the local cache. Hence, data in the system become inconsistent, which leads to an incorrect execution of a program. Thus, a mechanism is required, which updates memory locations in the system to keep memory consistent.

Coherency protocols are used to maintain memory coherency according to a given consistency model. Cache coherency may be maintained by hardware or software or a combination of the two. Tinuso currently does not implement a coherency mechanism in hardware, which makes it difficult to write programs and may limit the performance. There exist a plethora of protocols to implement cache coherent multicore systems. However, they are very difficult to implement, costly in hardware, and hard to scale [5]. In this chapter, we therefore

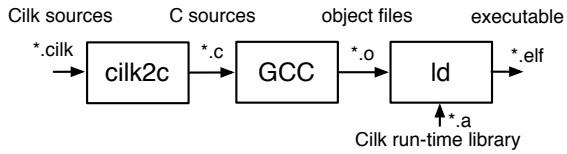


Figure 8.1: Cilk toolchain overview

consider simple restrictions in the programming model that allow for a simpler consistency model and enable a lightweight cache coherency mechanism.

8.2 Related Work

The need for simple and efficient parallel programming models has led to recent research interest in task-based models. Task-based programming models have been existing for a long time. For example, Cilk is a parallel multithreaded parallel programming language that has been developed since 1994 at the Massachusetts Institute of Technology, *MIT*. Several iterations on the language and the implementation of the runtime system have been developed and published until today [59]. Cilk is available as Open Source software whereas Cilk Plus is a commercial implementation by Intel [45]. The Cilk language is based on ANSI C and a few specific keywords to express task and data parallelism. Cilk is implemented as a source-to-source translator, *cilk2c*, which translates Cilk code into regular C code. The C code is then compiled with GCC and linked with the Cilk system runtime library to generate binary executables. The complete Cilk toolchain is illustrated in Figure 8.1.

The programmer expresses parallelism with Cilk-specific keywords such as *spawn* to create parallel tasks and *sync* to join parallel tasks [44]. The runtime-system decides which task is executed on which processor core. In Cilk, each processor core, or hardware thread, maintains a double-ended ready queue, *deque*, to store tasks that need to be processed. Processor cores, referred as workers, use the deque as a stack to push and pop tasks on to the end of the queue. When workers have an empty task queue, they become thieves and attempt to steal tasks from the top of another worker's deque, called victim. Cilk applies a work first principle where whenever a task is created, the parent task is suspended and the child task is executed. Suspended tasks are put on the deque and might get stolen by another processor core. Workers always operate on the

most recent created task to exploit locality. If the number of parallel tasks is much higher than the available processor cores, Cilk's work-stealing scheduling algorithm leads to an optimal load balance.

Cilk implements two versions of program code for each task: a fast clone, and a slow clone. The fast clone is used when a task is executed locally, hence it does not include overhead to support parallelism. The slow clone is used when a task is stolen and therefore requires full support for parallelism.

As workers and thieves operate on the same deque race conditions where thieves and the victim try to access the same task frame must be avoided. As the implementation of the work-stealing is performance critical, Cilk attempts to keep costs for the local worker as low as possible. Therefore Cilk only uses hardware locking primitives to arbitrate among different thieves and uses a protocol to detect situations where a thief and a victim access the same task. This is implemented with indices to the head and the tail of the deque. As thieves and workers access the deque, these indices are updated. The protocol is referred as THE protocol that allows for easily detecting deque anomalies if the head index is larger than the tail index. Once a race condition is detected, worker and thief roll-back the task execution and the local worker restart the task. Hence, the THE protocol removes overhead in the common case where there is no conflict. It is therefore 25% faster than an implementation with hardware locking primitives [45]. However, as these indices are stored in shared memory a sequential consistency or memory fence is required for the correct execution of the protocol.

Blumofe et. al. introduced a relaxed consistency model for Cilk-3 on the Connection Machine CM5 [16, 75]. The proposed model applies consistency at high-level of abstraction that corresponds to the task tree of a program, referred as directed acyclic graph, *DAG*. Cilk's implementation of DAG consistency is applied in software, using a page level granularity to minimize the overhead of managing shared objects.

There have been attempts to improve the scalability of Cilk with architectural support. Lont et. al. propose a solution where the programmer annotates shared memory addresses and architectural support is used for efficient memory consistency [77]. They use a consistency model with regions, which need to be annotated [55].

Just as Cilk, we aim for a fork-join task model for Tinuso to easily express parallelism and a runtime system to efficiently execute parallel programs. Cilk performs well on symmetric shared memory systems with a few processor cores. In contrast to Cilk, a programming language for Tinuso has to support heterogeneous multicore architectures. There have been extensions proposed to operate

Cilk on heterogeneous multicore platforms such as the Cell BE [98]. However, Tinuso systems may include a high number of heterogeneous cores and a global shared memory. Therefore, these proposed extension are not suitable for Tinuso systems.

OpenMP is a programming model defined by a group of major computer hardware and software vendors to support parallel programming in C/C++ and Fortran on shared-memory systems. OpenMP uses directives and associated clauses inserted into the source code to express the program's parallelism and data sharing. OpenMP directives use the `#pragma` mechanism provided by the C and C++ standards. Moreover, OpenMP provides a set of environment variables and library routines to interact with the runtime system. Production compilers such as GCC, and Intel Fortran and C/C++ compilers can be used to compile OpenMP programs. OpenMP is designed in a way that a program with all directives removed behaves as a valid sequential program. OpenMP was initially designed for developing parallel applications on platforms such as desktop computers and super-computers. Chapman et. al. identified limitations that prevent OpenMP to be used on embedded multicore systems [23]. For example, overheads for synchronization, task management, and memory consistency may be too high for embedded systems. Moreover, OpenMP v3.0 is targeted towards homogeneous multicore architectures with a global address space [17]. However, to support heterogeneous systems it is required that parts of a program can be bound to dedicated processor cores.

Recently, several OpenMP implementations for heterogeneous and embedded multicore systems have been proposed [23, 81]. In July 2013, updated OpenMP specifications got published that include support for heterogeneous systems [18]. For example, OpenMP 4.0 implements a new memory model and adds mechanisms to describe code sections where data and computation should be bound to specific computing devices. This new version of OpenMP also comes with enhancements in the task model that allow for aborting executing tasks, which is useful in search algorithms. Moreover, task dependencies can be expressed to synchronize tasks in streaming applications. However, OpenMP 4.0 needs to be implemented first. At the time compilers do not support OpenMP 4.0 yet.

OpenMP is emerging as a viable high-level programming model for parallel embedded systems. Different attempts have been made to map OpenMP directly to hardware, which enables offloading of tasks to reconfigurable devices [20, 76].

StarSs is another task-based programming model [95, 96]. It consists of few OpenMP-like pragmas that allow the programmer to express code segments that can be executed as parallel tasks. In contrast to Cilk, the programmer also

needs to identify inputs and outputs of tasks. This data is used by the runtime system to resolve dependencies and to schedule tasks to processor cores.

Etison et. al. take advantage of the input and output declaration used in StarSs to track task dependencies in hardware for determining when a task can be executed [41]. Task SuperScalar, is an abstraction of an out-of-order pipeline that operates at the task level. In the proposed out-of-order task pipeline, processor cores are considered as execution units and centralized memory structures implement reservation stations and renaming tables. Tasks are decoded and buffered in reservation stations. Once all operands are ready, a task is executed. Whenever a task finishes, all its data consumers in the reservation stations obtain a data ready signal to continue task execution. The proposed design efficiently uncovers task-level parallelism among tasks and achieves significant speedups over the StarSs software runtime.

Schedulers implemented in software can implement various scheduling algorithms that are optimized for the characteristics of a given application and programming model. For fine-grained task level parallelism, software scheduling implies significant overheads to synchronize and communicate task information. As this can be a limiting factor for the scalability of a multicore system, hardware scheduling has been proposed.

Carbon is a hardware technique to accelerate dynamic task scheduling [66]. Carbon implements hardware queues and a custom messaging protocol schedule tasks. A distributed task stealing scheduling scheme is applied to obtain a good cache performance. A set of extensions to the instruction set architecture is introduced to enable communication between processor pipeline and task queues. Carbon implements a global task unit to hold enqueued tasks in hardware queues for each thread. The advantage of this centralized design is that work stealing can be implemented efficiently by moving tasks between queues. Small local task units prefetch tasks to hide access latencies from the global task unit. Carbon's task stealing scheduler implements an efficient load balance algorithm, which leads to a speedup of 2.2x over software schedulers on a system with 128 threads.

Although Carbon enables significant speedups over software schedulers, it comes with the disadvantage of a fixed scheduling scheme. For example, it is not possible to adjust the stealing policy to improve locality. Sanchez et. al. therefore proposed a combined hardware-software approach to build fine-grained schedulers [101]. Simple architectural extensions allow for exchanging asynchronous messages between threads. This communication infrastructure provides a simple and flexible support to accelerate software schedulers for fine-grained parallelism. In contrast to Carbon, they implement task queues in soft-

ware. Their task queues perform efficiently only a single thread accesses them. Asynchronous direct messages, *ADM*, are used to implement task stealing without coherence traffic or synchronization overhead. This flexible approach allows for adapting scheduling to the characteristics of an application. Thus, ADM enables performance improvements of up to 70% over a Carbon implementation.

Hardware support to implement efficient fine-grained task scheduling is highly relevant for Tinuso. Computing systems on FPGA are very well suited to implement dedicated hardware primitives to accelerate the program execution because both hardware and software can be optimized for a given application.

Cache coherence protocols are designed to maintain the consistency between caches in multicore systems. A plethora of cache coherence protocols exist that can be classified as snoopy or directory based protocols. Snoopy protocols, maintain the shared status of each cache block. Cache controllers typically "snoop" on the bus and monitor transactions. If one of the cached blocks is involved in a bus transaction, it updates the local state information. Snoopy protocols are easy to implement when the communication infrastructure is a shared medium such as a bus. However, in on-chip networks it becomes more difficult to implement snoop mechanisms and the scalability is limited. Alternatively, directory-based cache coherency protocols maintain the state of memory blocks in a central directory. This directory includes a list of all processors that keep a copy of a cache block. Coherence is maintained as this directory is included in all memory transactions. However, directory-based cache coherency protocols are very complex and difficult to scale [5]. Moreover, cache coherency may incur a large hardware overhead and significantly increase the power consumption of a system. Hence, it is an active area of research to improve the scalability and power efficiency of cache coherence protocols [13, 19, 86, 123, 124].

Martin et. al. have investigated the hardware cache coherency as used in the Intel I7 processor [83]. They created a model to research scaling issues for network traffic, storage, energy consumption, maintaining inclusion, and latency. They conclude that the price for hardware cache coherency is marginal and does not justify to move complexity over to software. However, their network model is based on point-to-point connection and therefore their conclusions may not be applicable systems with a high number of cores. Multicore systems with a high number of cores are likely to employ a mesh based interconnect that infers additional latency per network hop. Hence the communication cost depends on the number of cores in the system.

DeNovo is a hardware-software co-design that seeks to simplify the hardware by enforcing disciplined parallel programming [25]. Denovo shows that using a deterministic programming model greatly simplifies the memory consistency

and enables more efficient communication and cache architectures. Denovo uses the Deterministic Parallel Java language, *DJP*, which is an extension of Java that adds deterministic parallel constructs. DJP enforces a deterministic programming model via compile-time type checking, which guarantees that parallel tasks are not interfering. The fact that parallel tasks do not have conflicting addresses allows DeNovo to reduce the number of states in the coherency protocol. The DeNovo protocol shows 15x fewer reachable states than a state-of-the-art implementation of the MESI protocol. The three stages of the protocol are: registered, valid, and invalid. While current state-of-the-art coherency protocols, such as MESI, require invalidation and acknowledge messages DeNovo implements a self-invalidation mechanism. The DeNovo compiler summarizes program regions into hardware regions and adds self-invalidation instructions. Hence, the software ensures that outdated data is invalidated. The evaluation of DeNovo shows better cache hit rates and network traffic than the MESI implementation, which translates to a better performance and energy savings.

Although deterministic programming is desirable for many applications, programs that contain non-deterministic parts are widespread, e.g. to implement synchronization locks. DeNovoND adds little overhead to DeNovo to support non-deterministic program code in the form of synchronization locks [110]. DeNovoND requires a signature per core, a bloom filter, and a LockQ state implement the memory consistency for locks. The evaluation of 11 benchmarks from SPLASH-2, PARSEC, and STAMP show that DeNovoND reduces the network traffic by 33% over a state-of-the-art MESI implementation.

Although Denovo greatly simplifies the cache coherency protocol there is still a directory required to maintain the status of cache blocks. If a central directory is coupled to the memory controller we can end up in situations where coherency traffic leads to contention in the network and reduces performance.

8.3 Semantics

A programming language for Tinuso multicore systems has to support task based programming to express parallelism in a simple way. As Tinuso multicore systems may be heterogeneous, a programming language has to allow for binding tasks to dedicated processing elements. Ideally, the programming language can make use of the existing GCC based Tinuso compiler infrastructure. As all these features are covered by the OpenMP specification 4.0, we decided to adopt some OpenMP directives for Tinuso to express parallel execution of programs.

In a first step, the programmer needs to define the available hardware. The

Listing 8.1: Simple example of an OpenMP task

```
1 /*export OMP_PLACES=(1,2,3,4,5,6,7,8);*/
2
3 void main(void)
4 {
5   #pragma omp task
6   printf ("Hello World");
7 }
```

OPM_PLACES environment variable is used to list identifiers of available processor cores in the system. Tinuso multicore systems are defined when the design is synthesized. When the proposed 2D mesh interconnect is used, processing elements are numbered with x and y coordinates. These hardware identifiers based on grid coordinates are used to implement lookahead routing. Hence, identifiers used to encode processing elements in OpenMP need to be mapped to the hardware identifiers to allow the runtime system to schedule tasks to the intended processing elements. This can be implemented in a system specification file. Listing 8.1 shows a simple example of a task being executed on a multicore system with eight processing elements. By default, the main function executes on the first processing element on the list. We define a task as an independent part of a program, which often is a call to C function. The `main` function spawns a task that prints "hello world". As nothing else is specified, this child task also runs on the same processing element as the parent task.

The programmer has the possibility to express parallelism by spawning tasks. To support heterogeneous architectures, the programmer can assign tasks to dedicated processing elements. Listing 8.2 illustrates an example where the `set_default_device` directive is used to bind a task, which computes the square root of a double precision floating point number to the processing element with the identifier 2. This is useful when some processor cores are equipped with dedicated floating point arithmetic hardware while others are not. Hence, it makes no sense that a core without hardware support for floating point arithmetic executes a task with a high number of floating point operations. We make the assumption that the programmer configures the system and therefore is aware of which processor cores are equipped with which functional units. The `taskwait` directive is used to wait for all children tasks to complete before execution on the parent task is continued.

Listing 8.2: Bind an OpenMP task to dedicated processing element

```
1  /*export OMP_PLACES=(1,2);*/
2
3  void foo(double a)
4  {
5    double x;
6    #pragma omp_set_default_device (2){
7      #pragma omp task
8      x = sqrt(a);
9      }
10   #pragma omp taskwait
11   ...
12 }
```

OpenMP allows for explicitly expressing tasks using the `task` directive. Alternatively, tasks may also be created implicitly with a `parallel` directive, which is described in Section 8.5.

8.4 Implementation

In large multicore systems memory bandwidth and latency are limiting factors. As we have seen in Chapter 7, the bandwidth of the memory controller limited the scalability of the direct back-projection algorithm on the Tinuso multicore system. In Chapter 4 we examined the impact of the memory latency where we compared the execution time of Tinuso core systems with and without on-chip interconnect. Moreover, we observed costly collision misses as functions become too large to fit into the instruction cache. For Tinuso, where on-chip memory is small, only a small fraction of a program can be held in caches. Hence we aim for a light-weight implementation of a runtime system, which attempts to keep the number of memory accesses as low as possible.

One performance critical part of the runtime system is the task queue. Cilk successfully implements a work-stealing algorithm to achieve a good load balancing in the system. However, when we implement the task queues in shared memory, we generate a lot of memory requests because processor cores execute tasks concurrently. Moreover, locking primitives or a protocol must be provided to handle race conditions where tasks are stolen from multiple processing ele-

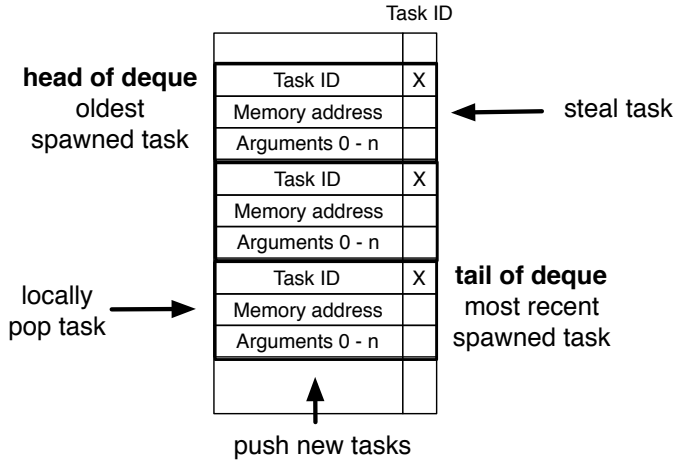


Figure 8.2: Overview of double ended task queue

ments. We therefore propose to implement the task queue in hardware. Block RAMs in FPGAs are well suited to implement queues. Each processing element shall maintain a doubly ended queue of ready tasks just as implemented in Cilk [45]. When tasks are spawned a task frame is put on the bottom of this queue. Tasks that are bound to a specific processor core do not go into this task queue. Instead, a task message is composed and sent to the destination core. A task message contains a task frame. For Tinuso, we define a task frame to consist of a task ID, a memory reference to the program code of the function and a list of function arguments. When a task is requested from another core, a frame is translated into a task message and transferred to the destination node and put on the bottom of the queue. Processor cores operate on the bottom of the local deque, pushing and popping tasks. Tasks are stolen from the head of the queue. Hence, the oldest tasks are stolen that likely exploit less locality than the newest tasks. When a task completes a return message is sent to the parent task. For Tinuso we limit it to include a single data type result only. It is up to the programmer to assure this. We avoid function arguments to be passed from one processing element to the other through the stack to attain lower memory traffic and to simplify the coherence mechanism. In contrast to Cilk we do not allow parent tasks to be stolen. This makes it easier for a child task to send the return message without the need of a broadcast.

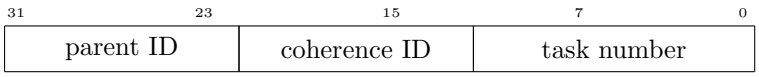


Figure 8.3: Overview of task ID

Figure 8.3 shows the encoding of the task ID. 10 bits are used to encode the processing element that spawns the task. 8 Bit of the 32 bit word are used to encode the coherence level while 14 bits are reserved for a task number.

8.4.1 Task Scheduling

OpenMP defines tasks as a specific instance of executable code and its data environment generated explicitly with a `task` directive or implicitly with a `parallel` directive [18]. However, it is the runtime system that takes scheduling decisions of how and when to execute a task. Commonly used task scheduling policies are breadth-first and work-first.

In breadth-first scheduling created tasks are put into a task queue while the parent task continues to create new tasks. Idle processor cores can grap tasks from the task queue and execute them. Contrary, in work-first scheduling, as implemented in Cilk, the parent task is suspended once a child task is created and the executing processor switches to child task. Suspended tasks are placed in a local task queue. Processor cores are working on tasks on their local queue, once it is empty they start looking for tasks to execute in the task queues of other processor cores. Work-first scheduling policy aims to follow the serial execution path of the algorithm to make better use of data locality.

Duran et. al. have investigated various scheduling policies for OpenMP tasks [37]. They obtain a better performance with work-first schedulers than with breadth-first schedulers. However, due to scheduling restrictions in OpenMP, they advocate for a breadth-first default scheduling policy. When a task completes it has to return to its parent task where tasks of a parallel region are joined. The proposed implementation of the runtime system envisions a local hardware task queue in each processing element. We do not want suspended parent tasks to be stolen, because it would result in costly broadcast messages when tasks have completed and have to return to their parent task. Hence, we cannot implement the work-first principle of Cilk. Instead we have to adopt the breadth-first scheduling.

In situations where a group of processor cores is assigned to execute a given parallel code region, the scheduler of the runtime system distributes a first set

of tasks to the available processor cores. Once completed tasks return, idle processor cores perform work-stealing until all tasks of a parallel code region are executed.

The task scheduling scheme is illustrated in Figure 8.5. On a system with four processor cores a program with two parallel regions is executed. It is assumed that the programmer did not assign tasks to dedicated processor cores. Hence, it is up to the scheduler in the runtime system to distribute the tasks among the four processor cores. For the first parallel region the parent task that runs on core 1, creates according to the breadth-first scheduling policy eight parallel tasks. The scheduler assigns the first three tasks to the other available processor cores in the system. The remaining five tasks of the first parallel region are put on the local task queue of core 1. Once the task on core 2 completes the result is sent back to core 1 and a task from the top of the queue is stolen. Core 3 and core 4 apply the same work-stealing policy, which is repeated until the task queue of core 1 is empty. While work-stealing is performed, the local core works on the bottom of the queue and executes tasks seven and eight.

As each task frame takes up space in the queue to encode the task ID, a memory reference to the program code and a list of arguments, it is possible that the queue overflows. Hence a mechanism is required that pauses task creation until there is space in the task queue again. We believe this to be a fair decision given the fact that breadth-first enables simple hardware implementation of the task queue, which reduces the number of memory access and avoids race conditions where multiple processing elements attempt to execute the same task.

8.4.2 Cache Coherence

The costs to implement a sequential consistency model are too high for a Tinuso system with a high number of processing elements. Instead, we aim to restrict the programming model and implement a very relaxed consistency model where memory becomes consistent only at the end of a parallel region. Between parallel tasks, memory is not kept coherent to reduce the number of data transfers. Instead, the system only triggers coherency actions when a task completes and writes modified data back to the memory controller. This implementation is scalable as no directory needs to be maintained and no invalidation messages need to be sent. While this implementation simplifies the cache coherency protocol, it also restricts the programmer in expressing parallelism. In a set of parallel tasks, there can only be either a single reader or writer of a memory location or multiple readers. It is up to the programmer to assure this. Figure 8.5

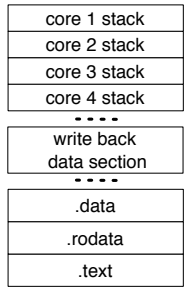


Figure 8.4: Memory sections overview.

shows a task graph with 2 parallel regions, memory only becomes consistent at the end of a parallel region. Each recursive function call is considered as a new parallel region and memory is updated after each iteration. The `taskwait` directive is used to mark the end of a parallel region where parallel tasks are joined. Before the program continues it must wait for the coherency actions to complete. Memory consistency can also be enforced manually with the `flush` directive, which triggers a write-back. To keep the number of write-backs low, each processor core only writes data located in a specific memory section back to main memory. This memory section is called `auto write-back` memory section. The programmer assigns data to this memory section with the `shared` directive. Figure 8.4 illustrates the envisioned memory space. The global shared memory space includes a text section for program code, a read only `rodata` section for constants and a section for data that will be modified during execution. The auto write-back data section is used to implement the described consistency model. Only modified data that resides in this section will be written back to main memory when a task completes and will be updated between parallel regions. Each processor maintains a private stack to store information about its active subroutines.

Figure 8.6 illustrates how the cache coherence is implemented. Whenever, data is fetched from the auto write-back memory section an ID consisting of an task ID of its parent and a coherence ID is stored. Cache tags in Tinuso implementation typically reside in block RAMs. The tag for a cache size of 4 kilo bytes with a burst length of 8 words with valid and dirty bit, takes up 20 bits.

As block RAMs come with a width of 36 bits, there are 16 bits left to encode an ID to maintain the coherency. This identifier may include parent

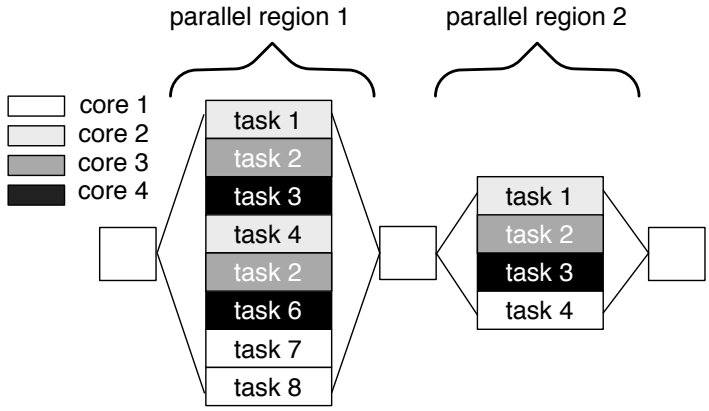


Figure 8.5: Task graph of a parallel multiplication on a Tinuso multicore system with 4 cores.

ID and the coherence ID. When data is in the cache is accessed, the ID of the current task is compared with the ID of the corresponding cache tag. If it matches, data is valid otherwise it needs to be re-fetched from main memory. This implementation ensures that within a parallel region data is not updated to avoid costly memory accesses while between successive parallel regions memory consistency is ensured. Modified data that resides in the auto write-back data section is marked with a modified bit and written back once a task finishes. Hence, each of the processor cores needs to know the upper and lower bound of the auto-write-back memory section and tag memory locations for automatic write-back.

It is intended to maintain cache coherency at cache line granularity, which requires alignment support in the compilation toolchain. However, cache coherency can be maintained at data word length with low overhead in hardware. An auto write-back status bit for each cached data word can be included in the block RAM that implements the data cache. When data is written back to main memory, a memory update message only includes the modified data words of a cache line. While this reduces memory traffic in some situations and does not require special memory alignment, it slightly increases the complexity of the memory controller, which has to ensure that the individual data words are written back to the correct memory address.

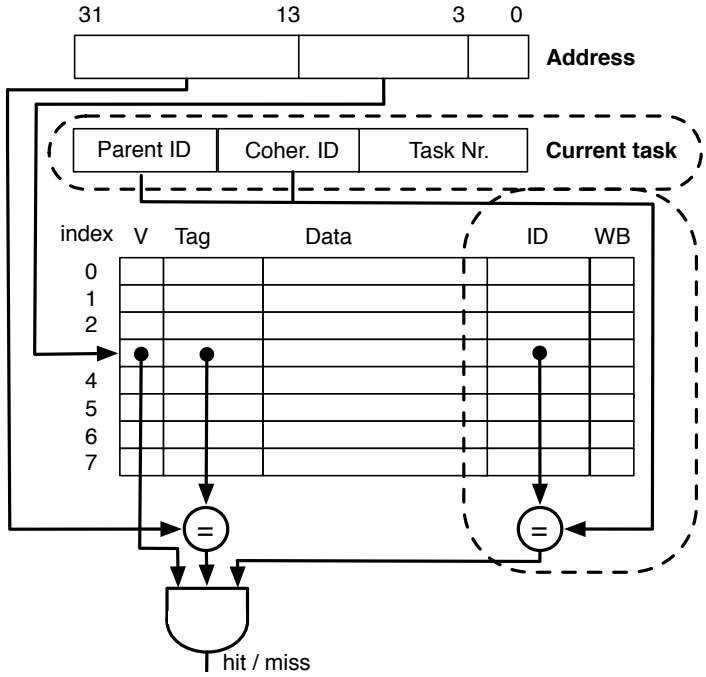


Figure 8.6: Tinuso extensions in the cache

8.5 Code Examples

To illustrate how to express parallelism, the following two subsections describe how to map a parallel matrix multiplication and the direct back-projection algorithm to Tinuso multicore systems, using OpenMP directives.

8.5.1 Parallel Matrix Multiplication

Figure 8.7 illustrates a parallel matrix multiplication [16]. The input matrices A and B are divided into four sub-matrices. The matrix multiplication is done by first computing products of the sub-matrices and second adding the results. As each of the eight multiplications is independent they can be executed in parallel. This leads to two intermediate result matrices, which need to be added to obtain the result of the matrix multiplication.

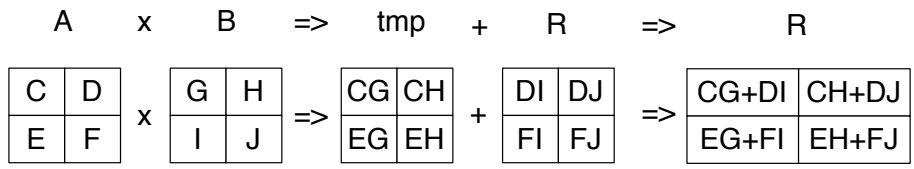


Figure 8.7: Matrix multiplication divide and conquer

The program code in Listing 8.3 shows how to express parallelism for a parallel matrix multiplication. Pointers to the input and result matrix are passed to the matrix multiplication function. Within this function, pointers to all of the sub-matrices are defined and assigned. For the given example we assume that input and result matrices are assigned to a shared memory section with automatic write-back functionality. Also the `tmp` matrix, which hold intermediate results is located in this memory section using the `shared` directive. When a task writes a memory location within this section, a coherency mechanism is triggered when the task completes that writes modified data back to main memory. In this example 8 processor cores are available. To execute the computation of the sub-matrices in parallel, the `parallel` directive is used. It creates, according to the clause, `num_threads(8)`, 8 parallel tasks. The `workshare` directive then divides the work among the available processor cores. Once, the sub-matrices are computed, four tasks are spawned, that perform the addition. When these four tasks are spawned, the `affinity(close)` clause is used to schedule these subtasks to processor cores that are close to the parent task. The implicit task creation with `parallel` and `workshare` directives gives the runtime system more freedom to map task to processing elements.

The algorithm in Figure 8.3 has one major drawback. When submatrices are computed, all results need to be written back to main memory to make sure that successive tasks perform the addition with the correct data. These successive tasks then need to re-fetch all data from memory, which is inefficient. Hence, if we can control, which task is executed on which core we can avoid this inefficiency and exploit locality. However, it requires that the `tmp` variable and the result matrix are not part of the shared memory section where automatic write-back is performed. The updated algorithm is shown in Listing 8.4. Binding tasks to dedicated processor cores may be beneficial to exploit locality but it may limit load balancing in the system. Hence, it depends on the application if this approach pays off.

Listing 8.3: Parallel matrix multiplication algorithm

```
1  /*export OMP_PLACES=(1,2,3,4,5,6,7,8);*/
2
3  void matmul(int *A, int *B, int *R)
4  {  /*assume result matrix R is shared*/
5     int *C, *D, *E, *F, *G, *H, *I, *J;
6     int *CG, *CH, *EH, *EG, *DI, *DJ, *FI, *FJ;
7     int tmp [n*n];
8     #pragma omp shared ( tmp )
9
10    /*get pointers to submatrices*/
11
12    /*parallel computation of submatrices*/
13    #pragma omp parallel num_threads(8){
14       #pragma omp workshare{
15       matrixmul(n,  C, G, CG);
16       matrixmul(n,  C, H, CH);
17       matrixmul(n,  E, G, EG);
18       matrixmul(n,  E, H, EH);
19       matrixmul(n,  D, I, DI);
20       matrixmul(n,  D, J, DJ);
21       matrixmul(n,  F, I, FI);
22       matrixmul(n,  F, J, FJ);}
23    }
24
25    /*parallel computation of result matrix*/
26    #pragma omp parallel num_threads(4)
27                         affinity(close){
28       #pragma omp workshare{
29         matrixadd(n,  CG, DI, R);
30         matrixadd(n,  CH, DJ, R);
31         matrixadd(n,  EG, FI, R);
32         matrixadd(n,  EH, FJ, R);}
33    }
34    return;
35   }
```

Listing 8.4: Parallel matrix multiplication algorithm with dedicated processor cores assigned to each task to exploit locality

```
1  /*export OMP_PLACES=(1,2,3,4);*/
2  void matmul(int *A, int *B, int *R)
3  {  /*assume result matrix is not shared*/
4     int *C, *D, *E, *F, *G, *H, *I, *J;
5     int *CG, *CH, *EH, *EG, *DI, *DJ, *FI, *FJ;
6     int tmp [n*n];
7
8     /*get pointers to submatrices*/
9
10    /*parallel computation of submatrices*/
11    #pragma omp_set_default_device (2){
12      #pragma omp task matrixmul(n,  C, G, CG);
13      #pragma omp task matrixmul(n,  D, I, DI);}
14    #pragma omp_set_default_device (3){
15      #pragma omp task matrixmul(n,  C, H, CH);
16      #pragma omp task matrixmul(n,  D, J, DJ);}
17    #pragma omp_set_default_device (4){
18      #pragma omp task matrixmul(n,  E, G, EG);
19      #pragma omp task matrixmul(n,  F, I, FI);}
20    #pragma omp_set_default_device (1){
21      #pragma omp task matrixmul(n,  E, H, EH);
22      #pragma omp task matrixmul(n,  F, J, FJ);}
23    }
24    /*parallel computation of result matrix*/
25    #pragma omp_set_default_device (2)
26      #pragma omp task matrixadd(n,  CG, DI, R);
27    #pragma omp_set_default_device (3)
28      #pragma omp task matrixadd(n,  CH, DJ, R);
29    #pragma omp_set_default_device (4)
30      #pragma omp task matrixadd(n,  EG, FI, R);
31    #pragma omp_set_default_device (1)
32      #pragma omp task matrixadd(n,  EH, FJ, R);
33    #pragma omp taskwait
34    return;
35   }
```

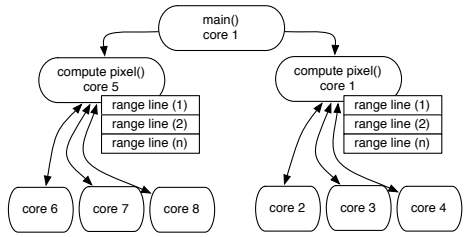


Figure 8.8: Example of work-stealing and push task scheduling

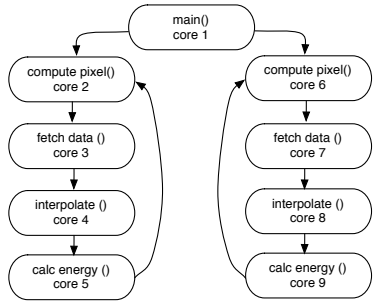


Figure 8.9: Example of pipelined task scheduling

8.5.2 SAR Direct Back-Projection Algorithm

The following examples implement the SAR direct back-projection algorithm described in Chapter 7. In the first example as shown in Figure 8.8, the task of computing a pixel is split up into two child tasks, which are bound to dedicated processor cores. These two child tasks then create three subtasks that each computes range line components. As shown in Listing 8.5 a `parallel for` directive is used to implicitly create tasks for each loop iteration. The clause `num_threads(3)` limits the runtime to create only 3 parallel subtasks at a time, while the clause `affinity(close)` ensures that these subtasks execute on processing cores close to the parent task.

As shown in the case study in Chapter 7, software pipelining can lead to a reduction in hardware resource usage. Tasks are split up in a number of subtasks that execute on dedicated processing elements, which are optimized for a given task.

Listing 8.5: Direct back-projection algorithm

```
1  /*export OMP_PLACES=(1,2,3,4,5,6,7,8);*/
2
3  int compute_range_line(i, j, n){
4    /*do computation*/
5    return result;
6  }
7
8  int compute_pixel(int i, int j){
9    /*500 range lines  to compute a single pixel*/
10   #pragma omp parallel for num_threads(3)
11                            affinity(close){
12     for (int n = 0; n < 500; n++) {
13       result += compute_range_line(i, j, n)
14     }
15   }
16   /*save or send pixel value*/
17   return 1;
18   }
19
20 void main(){
21   do {
22        /*calculate 1 pixel row*/
23        for (int i = 0; i < 1000; i++) {
24          /*calculate 1 pixel row*/
25          #pragma omp parallel num_threads(2)
26                                  affinity(spread){
27          for (int j = 0; j < 500; j++) {
28             compute_pixel(i, j);
29             compute_pixel(i, j+500);
30          }
31        }
32      }
33      /*save image and sync with input data*/
34      }
35    while (0);
36   }
```

Figure 8.9 shows a task graph of a software pipelined SAR back-projection. In a first step, two tasks are spawned that each compute half of the pixels of an image. The computation of a pixel is decomposed into three subtasks. First, the addresses of the relevant data for a given pixel are calculated and data is fetched. Second, an interpolation approximates the energy level for a given point on the ground. Third, amplitude and phase correction is applied. Listing 8.6 shows the source code of this algorithm. In this example the `get_default_device` function is used to determine on which processing element a certain task is executed. This is used to bind child tasks to neighboring processing elements. Software pipelining is achieved, as the `taskwait` directive is only applied to the first tasks. Thus the program waits only for the child tasks to complete, it does not wait for all descendants to finish. Hence, a new set of tasks is spawned as soon as the first subtask completes.

However, this may be a risky way to implement an application since task creation is synchronized with the first subtask only. If the subtasks later in the pipeline have a longer execution time than the first one, their task queue may overflow.

Listing 8.6: Software pipelined direct back-projection algorithm

```
1  /*export OMP_PLACES=(1,2,3,4,5,6,7,8,9);*/
2
3  calc_range_line(i, j, n, data){
4    /*calc_energy component and save result*/
5    return;
6    }
7
8  int interp_range_line(i, j, n, data){
9    int core = omp_get_default_device()
10   /*interpolate data*/
11   #pragma omp_set_default_device (core+1)
12   #pragma omp task calc_range_line(i,j,n,data);
13   return;
14   }
15
16 int fetch_range_line(int i, int j, int n){
17   int core = omp_get_default_device()
18   /*compute memory addresses and fetch data*/
19   #pragma omp_set_default_device (core+1)
20   #pragma omp task
```

```
21    interp_range_line(i,j,n,data);
22    return;
23    }
24
25  int compute_pixel(int i, int j){
26    int core = omp_get_default_device()
27    /*500 range lines to compute a single pixel*/
28    for (int n = 0; n < 500; n++) {
29        #pragma omp_set_default_device (core+1)
30        #pragma omp task{fetch_range_line(i,j,n)}
31        #pragma taskwait
32      }
33    /*save or send pixel value*/
34    return;
35    }
36
37  int main(){
38    do {
39       /*calculate 1 pixel row*/
40       for (int i = 0; i < 1000; i++) {
41         /*calculate 1 pixel row*/
42         for (int j = 0; j < 500; j++) {
43           #pragma omp_set_default_device (2)
44           #pragma omp task
45           compute_pixel(i, j);
46           #pragma omp_set_default_device (6)
47           #pragma omp task
48           compute_pixel(i, j+500);
49           #pragma omp taskwait}
50         }
51       /*save image and sync with input data*/
52       }
53     while (0);
54    }
```

8.6 Costs

We aim for a lightweight cache coherency mechanism and efficiently use hardware primitives to implement the runtime system. The costs for implementing the cache coherency mechanism are very low, identifiers to mark parallel regions are included in the block RAM that maintain cache tags. The additional status bits that mark modified data words can be included in the block RAMs that implement the data cache. Additional costs occur in registers to store the current task ID, the upper and the lower bound of the auto write-back memory section and the compare logic that comes with it. Moreover, the cache controller state machine needs to be extended to support the automatic write-back functionality. This functionality can either be implemented entirely in hardware or in a hardware-software co-design. The final implementation will show if it pays off to invest in hardware resources to reduce the penalty of a software routine that controls the automatic write back of modified data.

Tinuso provides infrastructure to maintain a relaxed memory consistency model. It is up to the programmer to keep the costs of the proposed coherency mechanism low. By storing as few data as possible in the automatic write-back data section the programmer avoids unnecessary data transfers. On the other hand, a conservative approach where a large portion of the data is put in the automatic write-back data section may execute slower but can be beneficial for debugging as it enforces a stronger memory consistency.

The hardware costs for implementing the task queue include a block RAM and logic elements for a state machine. This state machine has to handle incoming requests from the network that attempt to steal tasks and it has to allow the local processor core to push and pop tasks to the queue. Moreover, it has to arbitrate in case of simultaneous requests from the network and the local processor core and it has to detect overflow situations and pause task creation until there is space in the task queue again.

The processor core communicates with the task queue through the special memory space using `mtfs` and `mtms` instructions. As a task frame consists of a task ID, a memory reference, and function arguments only, it is very simple to create tasks and push them on the queue or to create a task message and send them dedicated processing elements. The existing network interface infrastructure is used to dispatch these task messages and to capture the results of returning task frames. Hence, the overhead to bind a task to a remote processing element is kept low. For example, a task message that calculates the square root of a double precision floating point number may consist of 4 data words only. One word is used for the task ID, one word encodes the memory

location of the program code and 2 data words are used for the double precision floating point argument. Task ID and memory address are constants that each are loaded with a `move-high` and `add-immediate` instruction pair and stored in the message buffer with a `mtms` instruction. Hence the overhead to create and dispatch a task message is about 10 clock cycles only if there are no cache misses. If we assume 10 clock cycles to transmit the task message to the destination core, 20 clock cycles overhead in the destination core to read in the task message and to create the result message, 10 clock cycles to transmit the result message and 10 clock cycles to read the result in, we obtain a total overhead of about 60 clock cycles. A hardware implementation of a double precision floating point operation may take up to 50 clock cycles while a pure software implementation of the same operation may take several hundreds of clock cycles. Hence, it pays off to bind fine-grained tasks to processing elements that include dedicated hardware support.

In a heterogeneous system, processing elements may be equipped with accelerators. Hence, a given software routine that benefits from these accelerators and is allowed to run on various processor cores may require multiple binary executables. Multiple implementations of a single software routine, runtime implementation, and task overheads leads to a larger program code. However, given the performance benefits and the low memory costs we believe this to be a fair decision.

8.7 Conclusions

Tinuso is a multicore system that may consist of hundreds of processor cores integrated in a single device. So far, the programmer had to be aware of many architectural details to successfully exploit the parallel architecture and orchestrate memory transfers, which is a difficult task. Therefore, a programming language, programming model, and runtime system are required to provide the programmer with a suitable abstraction of the underlying computer system and efficiently execute parallel programs. A fork-join task programming model was proposed to express parallelism and to enable a simple memory consistency model. We proposed to use OpenMP directives to annotate parallel regions of an application. Parallel tasks then are distributed among parallel processor cores. As Tinuso multicore systems may be heterogeneous, it must be possible to bind task to dedicated processor cores. It is also possible to bind tasks of a parallel region to a group of processor cores. The runtime system then performs a work-stealing scheduling policy to achieve an optimal load balance.

As a sequential memory consistency in large multicore systems is costly, Tinuso implements a relaxed consistency model where memory between parallel tasks is not kept coherent. Once a task finishes, coherency actions are triggered. Memory therefore only becomes consistent at the end of a parallel region. A simple restriction in the programming model ensures the correct execution of the program: within a parallel region there may only be one writer or multiple readers of a memory location.

Hardware support to implement a cache coherency mechanism was proposed. Cache lines are annotated with a coherency identifier, which ensures data to be updated between two parallel regions. Moreover, an automatic write-back of modified data was proposed to keep memory consistent. To keep the overhead of these coherency actions low, they only consider data that is manually placed in a dedicated shared memory section. Thus, the programmer is responsible for avoiding unnecessary data transfers.

Hardware primitives were proposed to support commonly used functionality of a programming model. As memory bandwidth is a critical resource in large multicore systems, we proposed to implement task queues in hardware. It reduces memory traffic and avoids data races when multiple cores access a task.

In a next step, the proposed runtime system and hardware primitives need to be implemented. There exists a benchmark suite that exploits OpenMP tasks, which can be used to evaluate the performance the proposed programming model and runtime system [38]. However, benchmarks need to be adapted to Tinuso's simplified memory consistency model.

Chapter 9

Conclusions

This thesis has addressed performance aspects of synthesizable computing systems. As the power efficiency and logic capacity of FPGAs increases, synthesizable computing systems become an attractive choice for embedded computing. However, a broad range of embedded applications demand high performance within severely constrained mechanical, power, and cost requirements. This thesis therefore researched architectural trade-offs to improve performance and programmability of synthesizable computing systems. A holistic approach was taken to optimize processor architecture, interconnection network, programming model, and runtime system for implementation on an FPGA. Addressing these design challenges has lead to four main research questions: 1) How to exploit instruction level parallelism that maps well on the internal structure of FPGAs. 2) To what extent are compilers able to leverage predicated execution and can this technique pay-off for synthesizable processors cores. 3) How to design scalable multicore systems and communication structures. 4) Define a runtime system and an abstraction of the computer system that allows the programmer to easily express parallelism and efficiently executes parallel programs.

- **Instruction level parallelism**: FPGAs consists of an array of configurable logic blocks that include lookup tables and successive flip-flops. Therefore the internal structure of an FPGA is well suited for pipelining. Complex multiported memory structures as used in superscalar machines, on the other hand, do not perform well on FPGA architectures. Therefore superpipelining was applied, which breaks pipeline stages into smaller stages and leverages instruction level parallelism by executing operations

partially overlapped. Results showed that pipelined caches, register file, and execution stage enable a significantly higher clock frequency but break the compatibility to existing instruction set architectures. Therefore, a new processor architecture, Tinuso, was designed to exploit current FPGA architectures. Tinuso is a lightweight architecture with a small instruction set that can easily be extended. Tinuso makes use of pipelined RAMs found in modern FPGAs to implement fast caches and register file. To keep the hardware design small, the pipeline is fully exposed to software where all types of hazards need to be considered. The architecture supports predicated execution to reduce the branch penalty. Tinuso takes advantage of its 8 stage pipeline that enables clock frequencies as high as 376 MHz on current state-of-the-art FPGAs. Tinuso optimally balances the logic between pipeline stages. The time critical path of the design includes only 4 successive 6-input lockup tables. Routing delays account for a substantial part of the critical path. Adding more pipeline stages is not beneficial as additional hardware resources and increasing complexity in the control logic lead to a diminishing return. The design was evaluated by running a set of numerical and search-based micro-benchmarks. Tinuso shows an average performance improvement of 38% over a similar Xilinx MicroBlaze configuration. Tinuso achieves a higher performance and consumes fewer hardware resources than commercial synthesizable processor cores.

- **Predicated execution**: The performance of a computer system highly depends on its efficiency to process control flow instructions. In a long processor pipeline where branch address and direction are resolved late in the pipeline branch instructions become costly. Tinuso therefore leverages predicated instructions to circumvent costly pipeline stalls. Predicated instructions allow for transforming control dependencies into data dependencies. Predicated instructions can therefore be used to replace simple if-statements. Moreover, Tinuso's predicated instructions can be used to fill delay slots. While predicated execution is conceptually simple, it is a challenge for a compiler to leverage predicated execution. We researched GCC's ability to exploit predicated execution. We concluded that GCC's if-conversion pass is too limited for Tinuso's predication scheme as only about 6% of the executed instructions make use of predicated execution. However, GCC successfully resolved hazards and was able to fill a large portion of Tinuso's delay slots. For a set of small C benchmarks Tinuso achieves an average speedup of 52% compared to a similar MicroBlaze

configuration. The encoding of predicated execution takes up 4 bits of each instruction word and leads to a complex forwarding logic. However, Tinuso's predication scheme enables very efficient, though hand-coded, assembly code. It allows for implementing highly efficient low-level libraries, which give Tinuso a performance advantage. Once the compiler makes better use of predicated execution the performance of Tinuso will be even higher.

- **Scalability**: The logic integration of modern FPGA's has reached a point where hundreds of Tinuso processor cores can be integrated on a single device. While it is conceptually trivial to compose multicore systems it remains a challenge to design a scalable system. The scalability of a system highly depends on the communication structures and the memory hierarchy. As there are limited on-chip memory resources on an FPGA processor cores often need to fetch data from main memory. It is therefore highly application dependent to which extent synthesizable computing systems scale. Efficient communication structures for Tinuso multicore systems were designed. A 2D mesh network topology was chosen as it maps well to the structure of FPGAs. The router architecture uses wormhole switching and a backpressure flow control mechanism to attain a latency of one clock cycle per hop. Routing scheme and flow control mechanism are optimized for high system clock frequency. Results showed that a pipelined feedback loop to manage contention leads to significantly higher clock speed and lower network latency at low injection rates. The scalability of Tinuso multicore systems was evaluated in terms of clock frequency. For a system with 48 cores, a maximum clock frequency of 300 MHz on a Xilinx Virtex 7 device was measured. Although synthesis tools reported a consistently high clock frequency for large-scale multicore system, tools that map multicore system to an FPGA have problems to efficiently map large multicore systems, which results in low system clock frequencies. Hence, substantial floor-planning support is required to attain a high system clock speed for large-scale systems. To demonstrate the scalable performance of Tinuso multicore systems a high performance radar data processing application was mapped to a system with 64 cores. This case study is based on the POLARIS synthetic aperture radar application, which requires real-time data processing for a 3000m wide area with a resolution of 2x2 meters. The multicore fabric consisting of 64 processor cores and 2D mesh network-on-chip utilizes 60% of the hardware resources of a Xilinx Virtex-7 device with 550 thousand logic cells and consumes about 10

watt only.

- **Programmability**: The radar data processing case study has shown that Tinuso multicore systems scale and deliver a high performance. However, the programmer needs to be aware of many architectural details to successfully exploit the parallel architecture and orchestrate memory accesses. This is a limiting factor for the productivity of a programmer. Therefore a runtime system and an abstraction of the computer system is required that allows the programmer to easily express parallelism and efficiently executes parallel programs. A fork-join task programming model was proposed to express parallelism. OpenMP directives are used to annotate parallel regions of an application. Parallel tasks then are distributed among available processor cores. As Tinuso multicore systems may be heterogeneous, it is supported to bind tasks to dedicated processor cores. As memory consistency in large multicore systems is costly, Tinuso implements a relaxed consistency model where memory between parallel tasks is not kept coherent. Once a task finishes, coherency actions are triggered. Main memory therefore only becomes consistent at the end of a parallel region. A simple restriction in the programming model, where there may only be a single writer or multiple readers to a memory location in a parallel region, ensures the correct execution of the program. Hardware support to implement a cache coherency mechanism was proposed. Cache lines are annotated with a coherency identifier, which ensures data to be updated between successive parallel regions. Moreover, an automatic write-back of modified data was proposed to keep memory consistent. To keep the overhead low these coherency actions only apply to data that are manually placed in a dedicated shared memory section. Hardware primitives were proposed to support commonly used functionality of a programming model. For example, a hardware implementation of task queues reduces memory traffic and avoids data races when multiple cores access a task.

The research for this thesis has led to the design of a synthesizable multicore system including compilation toolchain and a proposal for a programming model. In a next step, the proposed runtime system and hardware primitives need to be implemented. Existing OpenMP task benchmark suites can be used to evaluate the performance the proposed programming model. However, benchmarks need to be adapted to Tinuso's simplified memory consistency model. It is also intended to improve the if-conversion pass in the compiler to make better use of predicated execution.

We conclude Tinuso multicore systems are scalable and deliver a high performance. Multicore systems raise the abstraction level for the application programmer without facing the current performance drawbacks of high-level synthesis. The proposed programming model enables a high programming productivity while the hardware primitives in the runtime system lead to an efficient execution of parallel programs.

We therefore see several markets where synthesizable multicore systems are beneficial. As the radar case study has shown, Tinuso is an attractive choice for embedded applications, which demand a high performance within severely constrained mechanical, power, and cost requirements. As FPGAs are reconfigurable, synthesizable multicore systems can be modified and updated during product lifetime, which gives these systems a major benefit over embedded systems consisting of discrete devices. Moreover, synthesizable multicore systems are an attractive choice for products with a long lifetime, for example medical equipment. The production of discrete microcontrollers is often discontinued after a few years, which makes it difficult for companies to provide maintenance services for their systems. Synthesizable computing systems do not face this problem in the same way. If the production of a certain FPGA device is discontinued, a synthesizable multicore system can be mapped to a device of another FPGA family.

The research described in this thesis is also highly relevant for discrete multicore designs. Although ASIC technology enables faster processor pipelines and router designs than on FPGAs, off-chip memory access latency, and memory bandwidth remain the same. Hardware designers therefore have to deal with similar scaling problems as described in this work. Tinuso multicore systems are an attractive platform for research on multicore systems and parallel programming and the proposed runtime system and coherency mechanism are technology independent approaches to improve the scalability of multicore systems.

Appendix A

Application Binary Interface

This chapter describes the Tinuso Application Binary Interface, *ABI*. The Tinuso GNU toolchain follows the conventions described in this document.

A.1 Data Representation

The architecture of Tinuso uses the big-endian byte ordering scheme and supports 8-bit byte, 16-bit half-word, 32-bit word, and 64-bit double-word data types as listed in Table A.1. The address space is limited to 32 bits. No virtual Memory Management Unit, *MMU*, is implemented at the time. There is no support for Position-Independent Code, *PIC*.

A.2 Register Usage Conventions

The Tinuso architecture comes with a register file with 128 registers. These registers are used for predicate registers, integer and float registers. There are special registers available that can be accessed by mtms and mfms operations.

Table A.1: ANSI C data types

C type	Size (bits)
char	8
short	16
int	32
long int	32
long long int	64
pointer	32
float	32
double	64
long double	64

Table A.2: Tinuso register overview

register	software name	use	saver
r0-r7	r0-r7	predicate register	caller-saved
r8-r15	r8-r15	return register	caller-saved
r16-r31	r16-r31	argument register	caller-saved
r32-r63	r32-r63	general purpose	callee-saved
r64-r120	r32-r63	general purpose	caller-saved
r121	r121	defines i/o port	caller-saved
r122	rtmp	temporary register	caller-saved
r123 -124	r123-r124	reserved	caller-saved
r125	fp	hard frame pointer	callee-saved
r126	sp	stack pointer	callee-saved
r127	r127	return address register	caller-saved

A.3 Stack Conventions

Each function has a frame on the run-time stack. This stack grows downward from stack-start address as defined in the linker script. The stack conventions used by the Tinuso tools are shown in Table A.3. The stack pointer always holds the value of the end of the latest allocated stack frame.

Table A.3: Stack frame

Position	Contents	Frame
fp + 4n ... fp + 0	parameter n ... parameter 0	previous
fp - 4 fp - 8 fp - 16	function variables	current
sp + 4	value of previous fp	
sp + 0	return address	
sp - 4 ... sp - 4n	for use by leaf function	future

A.3.1 Calling Convention

The caller function passes parameters to the callee function using either the argument registers (r16-r31) or on the stack frame allocated for the caller function. Arguments of structures and unions are passed as pointers. Functions return results in the return registers (r8-r15). The GCC backend currently only uses r8 to return single word values. Double words values are returned in r8 and r9.

A.3.2 Machine Specific Registers

Tinuso may be configured to use machine specific registers. These registers are used for a broad range of applications such as communication to interfaces, capturing profiling data, and to readout processor specific information. An 32-bit indirect addressing scheme is used to access machine specific registers. Move To Machine Specific,*mtms*, and Move from Machine Specific,*mfms*, instructions move data from the register file to machine specific registers and vice versa. Table A.4 lists a number of registers used for this thesis.

A.4 ELF File Format

Executables are created by concatenating sections from the object files together and resolving symbolic references in these files. The Tinuso toolchain generates

Table A.4: Overview of Machine Specific Registers

Reg. Address	purpose
0x00000000	core id
0x00000100	profile control signals
0x0000101 ... 0x000010F	performance counters
0x00008000	uart tx
0x00008001	uart rx
	communication interface
0x00008000	read packet ready
0x00008001	read packet header
0x00008002	read packet data
0x00008003	write packet data
0x00008004	sent packet

executables in ELF object file format. Executables for the Tinuso architecture are marked with the unofficial machine number 0x1701 in the ELF header.

A.4.1 ELF File Sections

The compiler creates code that is split up in sections. Sections are marked with a number of flags that for example indicate whether the data in a section can be overwritten or not. According to these flags it is possible identify read-only sections of an executable and place them in a read only memory. Tinuso currently supports standard sections of object files as shown in Table A.5.

Table A.5: Section overview of an object or executable files

.text	text section
.rodata	read only data section
.data	read write data section
.bss	uninitialized data section

Appendix B

Instruction reference

add

Arithmetic Addition

[(!)rP] add rC,rA,rB

31 30 29 28	27 26 25 24 23	22	21 20 19 18	17 16 15 14 13	12 11 10 9 8	7 6 5 4	3 2 1 0
OP	rA	N	Pred	rC	rB		Funct
000	rA	N	Pred	rC	rB		0000

Description

Arithmetic addition of the register contents rA and rB; storing the result in register rC.

Pseudo-code

(rC) $\leftarrow$ (rA) + (rB)

addi

Arithmetic Addition Immediate

[(!)rP] addi rC,rA,Imm

31 30 29 28 27 26 25 24 23 22 21	20 19 18	17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
OP rA N	Pred	rC Immediate
001 rA N	Pred	rC Immediate

Description

Arithmetic addition of the register content rA and a 11-bit signed immediate; storing the result in register rC.

Pseudo-code

(rC) ← (rA) + Imm

and

Logical AND

[(!)rP] and rC,rA,rB

31 30 29 28	27 26 25 24 23	22	21 20 19 18	17 16 15 14 13	12 11 10 9 8	7 6 5 4 3	2 1 0
OP	rA	N	Pred	rC	rB	Funct	
000	rA	N	Pred	rC	rB	1000	

Description

Bitwise AND of the register contents rA and rB; storing the result in register rC.

Pseudo-code

(rC) $\leftarrow$ (rA) $\wedge$ (rB)

bnz

Branch on Not Zero

bnz rA, offset

31 30 29 28 27 26 25 24	23 22 21	20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
OP	rA	N	Reserved	Offset
010	rA	0	Reserved	Offset

Description

The offset value is shifted left two bits, sign-extended to program counter width, and then added to the PC of the branch instruction. The result is the effective branch address. If the content of register rA is not zero the program branches to target address with a delay of four instructions.

Pseudo-code

If (rA) $\neq$ 0 then
    PC $\leftarrow$ PC + exts(offset $\ll$ 2)
else
    PC $\leftarrow$ PC + 4

Scheduling Restrictions

bnz has 4 branch delay slots.

bsll

Barrel Shift Left Logical

[(!)rP] bsll rC, rA, rB

31 30 29 28 27 26 25 24	23 22 21	20 19 18 17 16	15 14 13 12 11	10 9 8 7 6 5 4 3 2 1 0
OP	rA	N	rP	rC	rB	Funct
111	rA	N	rP	rC	rB	0000

Description

Logically left shifts the content of register rA by the amount specified in register rB. The result stored in the register rC.

Pseudo-code

(rC) $\leftarrow$ (rA) $\ll$ (rB)[0:6]

Scheduling Restrictions

bsll has 3 delay slots.

Comments

bsll is an optional instruction and only valid if the target architecture has the barrel shift primitives enabled.

bsra — Barrel Shift Right Arithmetic

[(!)rP] bsra rC, rA, rB

31 30 29 28 27 26 25 24 23 22 21	20 19 18 17 16 15 14 13 12 11 10	9 8 7 6 5 4 3 2 1 0
OP rA N	rP rC	rB Funct
111 rA N	rP rC	rB 0001

Description

Arithmetically right shifts the content of register rA by the amount specified in register rB. The result stored in the register rC.

Pseudo-code

```
if ((rB)[0:6]) ≠ 0 then
    (rC)[32-(rB)[0:6]:31] ← (rA)[31]
    (rC)[0:31-(rB)[0:6]] ← (rA) ≪ (rB)[0:6]
else
```

Scheduling Restrictions

bsra has 3 delay slots.

bsrl

Barrel Shift Right Logic

[(!)rP] bsrl rC, rA, rB

31 30 29 28 27 26 25 24	23 22 21	20 19 18 17 16	15 14 13 12 11	10 9 8 7 6	5 4 3 2 1	0
OP	rA	N	rP	rC	rB	Funct
111	rA	N	rP	rC	rB	0010

Description

Logically right shifts the content of register rA by the amount specified in register rB. The result stored in the register rC.

Pseudo-code

(rC) $\leftarrow$ (rA) $\gg$ (rB)[0:6]

Scheduling Restrictions

bsrl has 3 delay slots.

Comments

bsrl is an optional instruction and only valid if the target architecture has the barrel shift primitives enabled.

bz

Branch on Zero

bz rA, offset

31 30 29 28 27 26 25 24 23 22 21	20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0			
OP	rA	N	Reserved	Offset
010	rA	1	Reserved	Offset

Description

The offset value is shifted left two bits, sign-extended to program counter width, and then added to the PC of the branch instruction. The result is the effective branch address. If the content of register rA is zero, the program branches to target address with a delay of four instructions.

Pseudo-code

if (rA) = 0 then
    PC $\leftarrow$ PC + exts(offset $\ll$ 2)
else
    PC $\leftarrow$ PC + 4

Scheduling Restrictions

bnz has 4 branch delay slots.

cmpseq Compare And Set Equal

[(!)rP] cmpseq rC, rA, rB

31 30 29 28 27 26 25 24 23 22 21	20 19 18 17 16 15 14 13 12 11 10	9 8 7 6 5 4 3 2 1 0

OP	rA	N	rP	rC	rB	Funct
000	rA	N	rP	rC	rB	0010

Description

If the contents of register rA and rB are equal rC is set to 1. Otherwise register rC is set to 0.

Pseudo-code

```
if (rA) = (rB) then
    (rC) ← 1
else
    (rC) ← 0
```

Scheduling Restrictions

cmpseq has 2 delay slots.
If the result of a **cmpseq** instruction is used for a branch instruction, there is 1 delay slot only.

cmpslt Compare And Set Less Than

[(!)rP] cmpseq rC, rA, rB

31 30 29 28 27 26 25 24 23 22	21	20 19 18 17	16 15 14 13 12 11 10	9 8 7 6 5 4 3 2 1 0
OP	N	rP	rC	rB	Funct
000	N	rP	rC	rB	0011

Description

Register rC is set to 1 if the content of register rA is less than the content of register rB. Otherwise register rC is set to 0.

Pseudo-code

```
if (rA) < (rB) then
    (rC) ← 1
else
    (rC) ← 0
```

Scheduling Restrictions

cmpseq has 2 delay slots.
If the result of a **cmpseq** instruction is used for a branch instruction, there is 1 delay slot only.

cmpsltu Compare And Set Less Than Unsigned

[(!)rP] cmpseq rC, rA, rB

31 30 29 28	27 26 25 24 23	22	21 20 19	18 17 16 15 14	13 12 11 10 9 8 7	6 5 4 3 2 1 0
OP	rA	N	rP	rC	rB	Funct
000	rA	N	rP	rC	rB	0100

Description

Register rC is set to 1 if the unsigned content of register rA is less than the unsigned content of register rB. Otherwise register rC is set to 0.

Pseudo-code

```
if (rA) < (rB) then
    (rC) ← 1
else
    (rC) ← 0
```

Scheduling Restrictions

cmpsltu has 2 delay slots.
If the result of a **cmpsltu** instruction is used for a branch instruction, there is 1 delay slot only.

jalr

Jump And Link Register

jalr rC,rA

31 30 29 28 27 26 25 24 23 22 21	20 19 18	17 16 15 14 13 12 11 10	9 8 7 6 5 4 3 2 1 0
OP rA N	rP	rC	Function
000 rA N	rP	rC	—'0001'1111

Description

The JALR instruction is an indirect absolute jump to a target address stored in rA with a delay of 4 clock cycles. The address of the instruction after the four delay slots is placed in rC (return address register).

Pseudo-code

PC ← rA
return_addr, PC ← rC + 20

Scheduling Restrictions

jalr has 4 delay slots.

ll

Load Linked

[(!)rP] ll rC,rA

31 30 29 28	27 26 25 24 23	22	21 20 19 18	17 16 15 14 13	12 11 10 9 8 7 6 5 4 3 2 1 0
OP	rA	N	rP	rC	Function
000	rA	N	rP	rC	—'0111'1111

Description

Loads a word from the word aligned memory location of registers rA and places the data in register rC. The memory location of a load-linked instruction is stored in the cache controller to track updates to this memory location. Load-linked is used in combination with a store-conditional instruction for synchronization in multithreaded programs.

Pseudo-code

Addr ← (rA) + offset
(rC) ← Mem(Addr)

Scheduling Restrictions

ll has 4 delay slots.

Comments

ll is an optional instruction and only valid if the target architecture has synchronization primitives enabled.

lw Load Word

[(!)rP] lw rC,rA,offset

31 30 29 28 27 26 25 24 23 22 21	20 19 18	17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
OP rA N	rP	rC Offset
100 rA N	rP	rC Offset

Description

Loads a word (32 bits) from the word aligned memory location that results from adding the contents of registers rA and the signed offset. The data is placed in register rC.

Pseudo-code

Addr ← (rM) + offset
(rC) ← Mem(Addr)

Scheduling Restrictions

lw has 4 delay slots.

mfms Move From Machine Specific

[(!)rP] mfms rC,rA

31 30 29 28	27 26 25 24 23	22	21 20 19 18 17	16 15 14 13 12	11 10 9 8 7 6 5 4 3 2 1 0
OP	rA	N	rP	rC	Function
000	rA	N	rP	rC	00000101111

Description

Loads a word (32 bits) from the special memory location where the content of register rA points to. The data is placed in register rC.

Pseudo-code

(rC) ← Special-Mem(rA)

Scheduling Restrictions

mfms has 2 delay slots.

movhi Move High Immediate

movhi rC,Imm

31 30 29 28 27 26 25 24	23 22 21 20 19 18 17 16	15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	
OP	Immediate	rC	Immediate
011	Immediate	rC	Immediate

Description

The signed 22-bit immediate value is shifted left by 10 bits, concatenated with zeros and stored in the register rC.

Pseudo-code

(rC) $\leftarrow$ (imm $\ll$ 10)

mtms Move To Machine Specific

[(!)rP] mtms rB, rA

31 30 29 28 27 26 25 24 23 22 21	20 19 18 17 16 15 14 13 12 11 10	9 8 7 6 5 4 3 2 1 0
OP rA N	rP rB	Func.
101 rA N	rP rB	11000000000

Description

Stores the contents of register rB, into the special memory location where the content of register rA points to.

Pseudo-code

Mem(rA) ← (rB)[31:0]

Scheduling Restrictions

mtms has 4 delay slots.

mul

Multiply

[(!)rP] mul rC,rA,rB

31 30 29 28 27 26 25 24 23 22 21	20 19 18 17	16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0	
OP	rA	N	Pred	rC	rB	Funct
111	rA	N	Pred	rC	rB	0011

Description

Multiplies the register contents rA and rB and stored the result in register rC. This 32-bit by 32-bit multiplication leads to a 64-bit result. The least significant word of the result is placed in rC, whereas the most significant word is discarded.

Pseudo-code

(rC) ← LSW((rA) x (rB))

Scheduling Restrictions

mul has 4 delay slots.

Comments

mul is an optional instruction and only valid if the target architecture has the multiplier primitive enabled.

or — Logical OR

[(!)rP] or rC,rA,rB

31 30 29 28 27 26 25 24 23 22 21	20 19 18 17 16 15 14 13 12 11 10	9 8 7 6 5 4 3 2 1 0
OP rA N	Pred rC	rA Funct
000 rA N	Pred rC	rA 1001

Description

Bitwise OR of the register contents rA and rB; storing the result in register rC.

Pseudo-code

(rC) $\leftarrow$ (rA) $\vee$ (rB)

sra — Shift Right Arithmetic

[(!)rP] sra rC,rA

31 30 29 28 27 26 25 24 23 22 21	20 19 18	17 16 15 14 13	12 11 10 9 8 7 6 5 4 3 2 1 0
OP	N Pred	rC	Function
010	rA N Pred	rC	—'1000'1111

Description

Shifts the register content rA to the right by one bit, storing the result in register rC. The empty position in the most significant bit is filled with a copy of the original MSB of rA.

Pseudo-code

(rC)[30:0] ← (rA)[31:1]
(rC)[31] ← (rA)[31]

srl Shift Right Logical

[(!)rP] srl rC,rA

31 30 29 28 27 26 25 24	23 22 21	20 19 18 17	16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
OP	rA	N Pred	rC Function
010	rA	N Pred	rC xxx'1001'1111

Description

Shifts the register content rA logically to the right by one bit, storing the result in register rC. A zero is placed in the most significant bit of rA.

Pseudo-code

(rC)[30:0] ← (rA)[31:1]
(rC)[31] ← 0

ll

Store Conditional

[(!)rP] ll rC,rA

31 30 29 28 27 26 25 24 23 22 21	20 19 18 17 16	15 14 13 12 11	10 9 8 7 6 5 4 3 2 1 0
OP	rA	N rP	rC	Function
000	rA	N rP	rC	—'0111'1111

Description

Store-conditional is used in combination with a load-link instruction. The store-conditional stores the content of register rB into the word aligned memory location of registers rA only if the memory location has not been updated since the load-link instruction. Load-link and store-conditional **ll/sc** are a pair of instructions used for synchronization in multithreaded programs.

Pseudo-code

Addr ← (rA) + offset
(rC) ← Mem(Addr)

Scheduling Restrictions

sc has 3 delay slots for load instructions to the same memory location (read-after-write hazard).

Comments

sc is an optional instruction and only valid if the target architecture has synchronization primitives enabled.

SW Store Word

[(!)rP] sw rB, rA, offset

31 30 29 28	27 26 25 24 23	22	21 20 19 18	17 16 15 14 13	12 11 10 9 8 7 6 5 4 3 2 1 0
OP	rA	N	rP	rB	Offset
101	rA	N	rP	rB	Offset

Description

Stores the contents of register rB, into the word aligned memory location that results from adding the content of registers rA and the signed offset.

Pseudo-code

Addr ← (rA) + offset
Mem(Addr) ← (rB)[31:0]

Scheduling Restrictions

sw has 3 delay slots for load instructions to the same memory location (read-after-write hazard).

sub

Arithmetic Subtraction

[(!)rP] sub rC,rA,rB

31 30 29 28 27 26 25 24 23 22 21	20 19 18 17	16 15 14 13 12 11 10	9 8 7 6 5 4 3 2 1 0
OP rA N	Pred	rC	rB Funct
000 rA N	Pred	rC	rB 0001

Description

Arithmetic subtraction of the register contents rA and rB; storing the result in register rC.

Pseudo-code

(rC) ← (rA) - (rB)

xor Logical XOR

[(!)rP] xor rC,rA,rB

31 30 29 28 27 26 25 24 23 22 21	20 19 18 17 16 15 14 13 12 11 10	9 8 7 6 5 4 3 2 1 0	
OP	rA	N	Pred	rC	rA	Funct
010	rA	N	Pred	rC	rA	1010

Description

Bitwise XOR of the register contents rA and rB; storing the result in register rC.

Pseudo-code

(rC) $\leftarrow$ (rA) $\otimes$ (rB)

Bibliography

[1] GCC back-end architecture listing. informational web page. gcc.gnu.org/backends.html. Retrieved on 16-August-2013.

[2] GCC front-end language listing. informational web page. gcc.gnu.org/frontends.html. Retrieved on 16-August-2013.

[3] Static single assignment book, working draft. ssa-book.gforge.inria.fr/latest/book.pdf, 2013. Retrieved on 16-August-2013.

[4] Ehliar A, P. Karlstrom, and D. Liu. A high performance microprocessor with DSP extensions optimized for the Virtex-4 FPGA. In *Proceedings of the 19th International Conference on Field Programmable Logic and Applications, FPL*, 2008.

[5] D. Abts, S. Scott, and D. Lilja. So many states, so little time: Verifying memory coherence in the cray x1. In *Proceedings of the International Parallel and Distributed Processing Symposium, IPDPS*, 2003.

[6] Adapteva. Adapteva E64G401 EPIPHANY 64-CORE MICROPROCESSOR Flyer. www.adapteva.com/wp-content/uploads/2013/06/e64g401_datasheet_4.13.6.14.pdf, 2013. Retrieved on 25-November-2013.

[7] Pritpal S. Ahuja, Pritpal S., Douglas W. Clark, and Anne Rogers. The performance impact of incomplete bypassing in processor pipelines. In *Proceedings of the 28th IEEE/ACM Annual International Symposium on Microarchitecture, MICRO*, 1995.

[8] Altera. Nios II processor reference handbook v10.1. www.altera.com/literature/hb/nios2/n2cpu_nii5v1.pdf, 2010. Retrieved on 25-March-2011.

[9] Altera. Handbook, v11.0. www.altera.com/literature/hb/nios2/n2cpu_nii, 2011. 5v1.pdf Retrieved on 8-August-2012.

[10] ARM. ARM Cortex-M1 Frequency and Area. www.arm.com/products/processors/cortex-m/cortex-m1.php. Retrieved on 5-October-2011.

[11] O. Azizi, A.Mahesri, B. Lee, S. Patel, and M. Horowitz. Energy-performance tradeoffs in processor architecture and circuit design: A marginal cost analysis. In *Proceedings of the 37th annual International Symposium on Computer Architecture, ISCA*, 2010.

[12] A. Baldassin, P. Centoducatte, and S. Rigo. An open-source binary utility generator. In *Journal ACM Transactions on Design Automation of Electronic Systems*, 2008.

[13] C. Ballapuram, A. Sharif, and H. Lee. Exploiting access semantics and program behavior to reduce snoop power in chip multiprocessors. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems ASPLOS*, 2008.

[14] J. Bennet. Howto: Porting newlib, a simple guide, application note 9, issue 1, 2010.

[15] M. Blom and P. Follo. VHF SAR image formation implemented on a gpu. In *Proceedings of the 25th IEEE International Geoscience and Remote Sensing Symposium, IGARSS'05*, 2005.

[16] R. Blumofe, M. Frigo, C. Joerg, C. Leiserson, and K. Randall. Dag-consistent distributed shared memory. In *Proceedings of the 10th Internationalarallel Parallel Processing Symposium, IPPS*, 1996.

[17] OpenMP Architecture Review Board. Openmp application program interface specification. In *Specification Version 3.0*, 2008.

[18] OpenMP Architecture Review Board. Openmp application program interface specification. In *Specification Version 4.0*, 2013.

[19] I. Burcea, J. Zebchuk, and A.Moshovos. Teaching old caches new tricks: Regiontracker and predictor virtualization. In *IEEE Conference on Communications, Computers and Signal Processing*, 2009.

[20] D. Cabrera, X. Martorell, G. Gaydadjiev, E. Ayguade, and D. Jimenez-Gonzalez. Openmp extensions for fpga accelerators. In *Proceedings of the International Symposium on Systems, Architectures, Modeling, and Simulation SAMOS*, 2009.

[21] W. Carrara, R. Goodman, and R. Majewski. *Spotlight Synthetic Aperture Radar: Signal Processing Algorithms*. Artech House, 1995.

[22] C. Casteel, J. Gorham, M. Minardi, S. Scarborough, K. Naidu, and U. Majumder. A challenge problem for 2d/3d imaging of targets from a volumetric data set in an urban environment. In *Proceedings of Algorithms for Synthetic Aperture Radar Imagery XIV*, 2007.

[23] B. Chapman, L. Huang, E. Biscondi, E. Stotzer, A. Shrivastava, and A. Gatherer. Implementing openmp on a high performance embedded multicore mpsoc. In *Proceedings of the IEEE International Symposium on Parallel and Distributed ProcessingIPDPS*, 2009.

[24] Derek Chiou, Huzefa Sunjeliwala, Dam Sunwoo, John Xu, and Nikhil Patil. FPGA-based Fast, cycle-accurate, full-system simulators. In *IEEE/ACM international conference on Computer-aided design, ICCAD*, 2007.

[25] B. Choi, R. Komuravelli, H. Sung, R. Smolinski, N. Honarmand, S. Adve, V. Adve, N. Carter, and C. Chou. DeNovo: Rethinking the memory hierarchy for disciplined parallelism. In *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems ASPLOS*, 2013.

[26] E. Christensen, N. Skou, J. Dall, K. Woelders, J. Jøorgensen, J. Granholm, and S. Madsen. EMISAR: an absolutely calibrated polarimetric l- and c-band SAR. 1998.

[27] Eric S. Chung, James C. Hoe, and Babak Falsafi. Protoflex: Co-simulation for component-wise fpga emulator development.

[28] Eric S. Chung, E. Nurvitadhi, James C. Hoe, Babak Falsafi, and Ken Mai. Virtualized full-system emulation of multiprocessors using FPGAs. *Workshop on Architectural Research Prototyping held in conjunction with the 34th International Symposium on Computer Architecture*, 2007.

[29] Eric S. Chung, E. Nurvitadhi, James C. Hoe, Babak Falsafi, and Ken Mai. A complexity-effective architecture for accelerating full-system multiprocessor simulations using FPGAs. *16th international ACM/SIGDA symposium on Field programmable gate arrays, FPGA*, 2008.

[30] B. Cordes and M. Leeser. Parallel backprojection: A case study in high-performance reconfigurable computing. In *Proceedings of the 14th IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM*, 2009.

[31] Compaq Computer Corporation. Compaq Computer Corporation Alpha Architecture Reference Manual 4th edition. download.majix.org/dec/alpha_arch_ref.pdf, 2002. Retrieved on 5-December-2013.

[32] S. Craven, C. Patterson, and P. Athanas. Configurable soft processor arrays using the openfire processor. In *Proceedings of the 8th Annual Conference on Military and Aerospace Programmable Logic Devices MAPLD*, 2005.

[33] D. Curd. Qdr ii sram interface for virtex-4 devices, v.2.4, xapp703. www.xilinx.com/support/documentation/application_notes/xapp703.pdf, 2008. Retrieved on 9-June-2012.

[34] J. Dall, J. Joergensen, E. Christensen, and S. Madsen. Real-time processor for the danish airborne SAR. In *IEE Proceedings-F, vol. 139*, 1992.

[35] J. Dall, S. Kristensen, V. Krozer, C. Hernandez, J. Vidkjær, A. Kusk, J. Balling, N. Skou, S. Søbjrg, and E. Christensen. ESA's polarimetric airborne radar ice sounder (POLARIS): design and first results. In *Journal on Radar, Sonar Navigation, IET, vol. 4*, 2010.

[36] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *Proceedings of the 25th annual International Symposium on Computer Architecture, ISCA*, 1998.

[37] A. Duran, J. Corbalan, and E. Ayguade. Evaluation of openmp task scheduling strategies. In *Proceedings of the 4th international conference on OpenMP in a new era of parallelism IWOMP*, 2008.

[38] A. Duran, X. Teruel, R. Ferrer, X. Martorell, and E. Ayguade. Barcelona openmp tasks suite: A set of benchmarks targeting the exploitation of task

parallelism in openmp. In *Proceedings of the International Conference on Parallel Processing, ICPP*, 2009.

[39] A. Ehliar and D. Liu. An ASIC perspective on fpga optimizations. In *Proceedings of the 20th International Conference on Field Programmable Logic and Applications FPL*, 2009.

[40] European Space Agency ESA. Measuring forest biomass from space - esa campaign tests biomass mission. In *www.esa.int/esaLP/SEMFCJ9RR1F_index_0.html*. Retrieved on 5-May 2012.

[41] Y. Etsion, F. Cabarcas, A. Rico, A. Ramirez, R. Badia, E. Ayguade, J. Labarta, and M. Valero. Task superscalar: An out-of-order task pipeline. In *Proceedings of the 43th IEEE/ACM Annual International Symposium on Microarchitecture, MICRO*, 2010.

[42] F. Mesa-Martinez et al. SCOORE santa cruz out-of-order RISC engine, FPGA design issues. In *Proceedings of Workshop on Architectural Research Prototyping WARP, held in conjunction with ISCA-33*, 2006.

[43] A. Fasih and T. Hartley. Gpu-accelerated synthetic aperture radar backprojection in cuda. In *Proceedings of the IEEE International Radar Conference*, 2010.

[44] Supercomputing Technologies Group MIT Laboratory for Computer Science. Cilk 5.4.6 reference manual. 1998.

[45] M. Frigo, C. Lerserson, and K. Randall. The implementation of the cilk-5 multithreaded language. In *Proceedings of the 19th ACM SIGPLAN Conference on Programming Language Design and Implementation PLDI*, 1998.

[46] T. Gingold. Ghdl, a vhdl compiler, v0.22. attila.kinali.ch/ghdl.pdf, 2006. Retrieved on 16-August-2013.

[47] S. Gorlatch. Send-receive considered harmful: Myths and realities of message passing. In *ACM Journal on Transactions on Programming Languages and Systems, vol. 26, issue 1*, 2004.

[48] Stanford Concurrent VLSI Architecture Group. Open source network-on-chip router RTL. In *nocs.stanford.edu/cgi-bin/trac.cgi/wiki/Resources/Router*. Retrieved on 26-May 2013.

[49] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET benchmarks – past, present and future. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis WCET*, 2010.

[50] C. Hansen and T. Riordan. Risc computer with unaligned reference handling and method for the same, US Patent 4814976, 1989.

[51] R. Hasti and S. Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation PLDI*, 1998.

[52] A. Hellmund. GCC frontend internals, v0.1. blog.lxgcc.net/wp-content/uploads/2011/03/GCC_frontend.pdf, 2011. Retrieved on 16-August-2013.

[53] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach, 4th ed.* Morgan Kaufmann Publishers, 2006.

[54] Y. Huan and A. DeHon. Fpga optimized packet-switched noc using split and merge primitives. In *Proceedings of International Conference on Field-Programmable Technology FPT*, 2012.

[55] L. Iftode, J. Singh, and K. Li. Scope consistency: a bridge between release consistency and entry consistency. In *Proceedings of the eighth annual ACM symposium on Parallel algorithms and architectures, SPAA*, 1996.

[56] Texas Instruments. TMS320C62x DSP CPU and Instruction Set Reference Guide, SPRU731A. www.ti.com/lit/ug/spru731a/spru731a.pdf, 2010. Retrieved on 8-October-2013.

[57] A. Janarthanan, V. Swaminathan, and K. Tomko. MoCReS: an Area-Efficient Multi-Clock On-Chip Network for Reconfigurable Systems. In *Proceedings of IEEE Computer Society Annual Symposium on VLSI, ISVLSI*, 2007.

[58] N. Jensen, P. Larsen, R. Ladelsky, A.Zaks, and S. Karlsson. Guiding programmers to higher memory performance. In *Proceedings of 5th Workshop on Programmability Issues for Heterogeneous Multicores MULTIPROG*, 2012.

[59] C. Joerg. The cilk system for parallel multi-threaded computing. In *PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology*, 1996.

[60] N. Jouppi and D. Wall. Available instruction-level parallelism for superscalar and superpipelined machines. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems ASPLOS*, 1989.

[61] Kalray. Kalray MPPA Flyer. www.kalray.eu/IMG/pdf/FLYER_MPPA_M, 2012. ANYCORE-3.pdf Retrieved on 25-November-2013.

[62] K.Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th annual International Symposium on Computer Architecture, ISCA*, 1990.

[63] J. Kim, S. Lee, S. Moon, and S. Kim. Comparison of llvm and gcc on the arm platform. In *Proceedings of the 5th International Conference on Embedded and Multimedia Computing EMC*, 2010.

[64] D. Koch, C. Haubelt, and J. Teich. Efficient reconfigurable on-chip buses for FPGAs. In *Proceedings of the 2008 16th International Symposium on Field-Programmable Custom Computing Machines, FCCM'08*, 2008.

[65] A. Krasnov, A. Schultz, J. Wawrzynek, G. Gibeling, and P. Droz. Ramp blue: a message-passing manycore system in fpgas. In *Proceedings of the International Conference on Field Programmable Logic and Applications, FPL*, 2007.

[66] S. Kumar, C. Hughes, and A. Nguyen. Carbon: architectural support for fine-grained parallelism on chip multiprocessors. In *Proceedings of the 34th annual International Symposium on Computer Architecture, ISCA*, 2007.

[67] T. Kumura, S. Taga, N. Ishiura, Y. Takeuchi, and M. Imai. Automatic generation of gnu binutils and gdb for custom processors based on plug-in method. In *Proceedings of the Workshop on Synthesis And System Integration of Mixed Information TechnologiesSASIMI*, 2012.

[68] A. Kusk and J. Dall. SAR focusing of p-band ice sounding data using back-projection. In *Proceedings of the 30th IEEE International Geoscience and Remote Sensing Symposium, IGARSS'10*, 2010.

[69] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. In *IEEE Transactions on Computers*, 1979.

[70] Lattice. LatticeMico32 Product Brief I0186. www.latticesemi.com/documents/I0186.pdf, 2007. Retrieved on 25-March-2011.

[71] Lattice. Latticemico32 processor reference manual volume 8.1. www.latticesemi.com/documents/lm32_archman.pdf, 2010. Retrieved on 25-March-2011.

[72] I. Lebedev, C. Shaoyi, A. Doupnik, J. Martin, C.Fletcher, D.Burke, L. Mingjie, and J. Wawrzynek. MARC: A many-core approach to reconfigurable computing. In *Proceedings of the International Conference on Reconfigurable Computing and FPGAs, ReConFig*, 2010.

[73] H. Lee, N. Chang, U. Ogras, and R. Marculescu. On-chip communication architecture exploration: A quantitative evaluation of point-to-point, bus, and network-on-chip approaches. volume 12, 2008.

[74] T. Lehmann, C. Goenner, and K. Spitzer. Survey: Interpolation methods in medical image processing. In *IEEE Journal on Transactions on Medical Imaging, vol. 18*, volume 18, 1999.

[75] C. Leiserson, Z. Abuhamdeh, D. Douglas, C. Feynman, M. Ganmukhi, J. Hill, D. Hillis, B. Kuszmaul, M. St. Pierre, D. Wells, M. Wong, S. Yang, and R. Zak. The network architecture of the connection machine cm-5. In *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures, SPAA*, 1992.

[76] Y. Leow, C. Ng, and W. Wong. Generating hardware from openmp programs. In *Proceedings of the IEEE International Conference on Field Programmable Technology,FPT*, 2006.

[77] G. Long, D. Fan, and J. Zhang. Architectural support for cilk computations on many-core architectures. In *ACM SIGPLAN Notices, vol.44*, 2009.

[78] Y. Lu, J. McCanny, and S.Sezer. Generic Low-Latency NoC Router Architecture for FPGA Computing Systems. In *Proceedings of the 21th International Conference on Field Programmable Logic and Applications, FPL*, 2011.

[79] S. Mahlke, R. Hank, J. McCormick, D. August, and W. Hwu. A comparison of full and partial predicated execution support for ilp processors. In *Proceedings of the 22th International Symposiom on Computer Architecture ISCA*, 1995.

[80] S. Mahlke, D. Lin, W. Chen, R. Hank, and R. Bringmann. Effective compiler support for predicated execution using the hyperblock, 1992.

[81] A. Marongiu, P. Burgio, and L. Benini. Supporting openmp on a multi-cluster embedded mpsoc. In *Journal Microprocessors and Microsystems, vol. 35, issue. 8*, 2011.

[82] J. Marrill. GENERIC and GIMPLE: A new tree representation for entire function. ftp://gcc.gnu.org/pub/gcc/summit/2003/GENERIC_and_GIMPLE.pdf, 2003. Retrieved on 16-August-2013.

[83] M. Martin, M. Hill, and D. Sorin. Why on-chip cache coherence is here to stay. In *Magazine on Communications of the ACM, vol. 55*, 2012.

[84] P. Minev and V. Stoianova Kukenska. The Virtex-5 Routing and Logic Architecture. In *Annual Journal of Electronics, ISSN 1313-1842*, 2009.

[85] H. Nilsson. Porting gcc for dunces. ftp.axis.com/pub/users/hp/pgccfd/pgccfd.pdf, 2000. Retrieved on 16-August-2013.

[86] J. Nilsson, A. Landin, and P. Stenström. The coherence predictor cache: A resource-efficient and accurate coherence prediction infrastructure. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing IPDPS*, 2003.

[87] G. Nychis, C. Fallin, T. Moscibroda, and O. Mutlu. Next generation on-chip networks: what kind of congestion control do we need? In *Proceedings of the Ninth ACM SIGCOMM Workshop on Hot Topics in Networks,HOTNETS'10*, 2010.

[88] IEEE Task P754. ANSI/IEEE 754-1985, standard for binary floating-point arithmetic. pages 1–18, 1985.

[89] S. Palacharla, N. Jouppi, and J. Smith. Complexity-effective superscalar processors. In *Proceedings of International Symposium on Computer Architecture ISCA-24*, 1997.

[90] A. Papakonstantinou, Y. Liang, J. Stratton, K. Gururaj, D. Chen, W. Hwu, and J. Cong. Multilevel granularity parallelism synthesis on fpgas. In *Proceedings of the 19th IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM*, 2011.

[91] M. Papamichael and J. Hoe. Connect: Re-examining conventional wisdom for designing nocs in the context of fpgas. In *Proceedings of International Symposium on Field-Programmable Gate Arrays FPGA*, 2012.

[92] J. Park and M. Schlansker. On predicated execution, 1991.

[93] D. Patterson. Reduced Instruction Set Computers. *ACM Journal Communication, vol. 28, no. 1*, 1985.

[94] D. Patterson and C. Sequin. A vlsi risc. In *Journal Computer, vol. 15, issue 9*, 1982.

[95] J. Perez, R. Badia, and J. Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *Proceedings of the International Conference on Cluster Computing CLUSTER*, 2008.

[96] J. Planas, R. Badia, E. Ayguadé, and J. Labarta. Hierarchical task-based programming with starss. In *International Journal of High Performance Computing Applications IJHPCA, vol. 23, no. 3*, 2009.

[97] M. Purnaprajna and P. Ienne. Making wide-issue VLIW processors viable on FPGAs. *ACM Journal Transactions on Architecture and Code Optimization (TACO), vol. 8, no. 33*, 2012.

[98] S. Rajopadhye and M. Strout. Cellcilk: Extending cilk for heterogeneous multicore platforms. In *Lecture Notes in Computer Science Languages and Compilers for Parallel Computing, vol. 7146*, 2013.

[99] J. Raygoza-Panduro, S. Ortega-Cisneros, J. Rivera, and A. de la Mora. Design of a mathematical unit in FPGA for the implementation of the control of a magnetic levitation system. In *International Journal of Reconfigurable Computing*, 2008.

[100] A. Roca, J. Flich, and G. Dimitrakopoulos. Desa: Distributed elastic switch architecture for efficient networks-on-fpgas. In *Proceedings of the 22nd International Conference on Field Programmable Logic and Applications FPL*, 2012.

[101] D. Sanchez, R. Yoo, and C. Kozyrakis. Flexible architectural support for fine-grain scheduling. In *Proceedings of the 15th international conference on Architectural support for programming languages and operating systems ASPLOS*, 2010.

[102] S. Scarborough, C. Casteel, J. Gorham, M. Minardi, U. Majumder, M. Judge, E. Zelnio, and M. Bryant. A challenge problem for sar-based gmti in urban environments. In *Proceedings of Algorithms for Synthetic Aperture Radar Imagery XVI*, 2009.

[103] P. Schleuniger and S. Karlsson. Tinuso: A processor architecture for a multi-core hardware simulation platform. In *Proceedings of third swedish workshop on Multi-Core Computing MCC*, 2010.

[104] P. Schleuniger, A. Kusk, J. Dall, and S. Karlsson. Synthetic aperture radar data processing on an fpga multi-core system. In *Proceedings of the 26th International Conference on Architecture of Computing Systems ARCS*, 2013.

[105] P. Schleuniger, S. A. McKee, and S. Karlsson. Design principles for synthesizable processor cores. In *Proceedings of the 25th International Conference on Architecture of Computing Systems ARCS*, 2012.

[106] B. Sethuraman, P. Bhattacharya, J. Khan, and R. Vemuri. LiPaR: A lightweight parallel router for FPGA-based networks-on-chip. In *Proceedings of the 15th ACM Great Lakes symposium on VLSI, GLSVLSI*, 2005.

[107] B. Sethuraman and R. Vemuri. Multi2 Router: A Novel Multi Local Port Router Architecture with Broadcast Facility for FPGA-Based Networks-on-Chip. In *Proceedings of International Conference on Field Programmable Logic and Applications FPL*, 2006.

[108] R. Stallman and the GCC Developer Community. GNU compiler collection internals, for gcc version 4.9.0 (pre-release). gcc.gnu.org/onlinedocs/gccint.pdf, 2013. Retrieved on 16-August-2013.

[109] W. Strecker. Vax-11/780: A virtual address extension to the dec pdp-11 family. In *Proceedings of the National Computer Conference*, 1978.

[110] H. Sung, R. Komuravelli, and V. Adve. DeNovoND: Efficient hardware support for disciplined non-determinism. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques PACT*, 2011.

[111] Tilera. TILEPro64 Product Brief I0186. www.tilera.com/sites/default/files/productbriefs/TILEPro64_Processor_, 2011. PB019_v4.pdf Retrieved on 25-November-2013.

[112] J. Tong, I.Anderson, and M. Khalid. Soft-core processors for embedded systems. *Proceedings of the International Conference on Microelectronics, ICM06*, 2006.

[113] L. Ulander, H. Hellsten, and G. Stenstroem. Synthetic-aperture radar processing using fast factorized back-projection. In *IEEE Journal on Transactions on Aerospace and Electronic Systems, vol. 39*, 2003.

[114] A. Waterman, Y. Lee, D. Patterson, and K. Asanovic. The RISC-V Instruction Set Manual version 1.0. inst.eecs.berkeley.edu/ cs152/sp12/handouts/riscv-spec.pdf, 2012. Retrieved on 5-December-2013.

[115] J. Wawrzynek, D. Patterson, M. Oskin, Shin-Lien Lu, C. Kozyrakin, J.C. Hoe, D. Chiou, and K. Asanovic. RAMP: Research accelerator for multiple processors. 2007.

[116] V. Weaver and S. A. McKee. Are cycle accurate simulations a waste of time? In *Proceedings of the 7th Workshop on Duplicating, Deconstructing, and Debunking*, 2008.

[117] C. Wu, K. Liu, and M. Jin. Modeling and a correlation algorithm for spaceborne sar signals. In *IEEE Journal on Transactions on Aerospace and Electronic Systems, vol. 18*, 1982.

[118] Xilinx. 7 series FPGAs overview DS180 v1.5. www.xilinx.com/support/documentation/data_sheets/ds180_7Series_ Overview.pdf, 2011. Retrieved on 25-March-2011.

[119] Xilinx. LogiCORE IP floating-point operator v5.0 ds335. www.xilinx.com/support/documentation/ip_d, 2011. ocumentation/floating_point_ds335.pdf Retrieved on 7-June-2012.

[120] Xilinx. MicroBlaze Processor Reference Guide UG081 volume 12.0. www.xilinx.com/support/documentation/sw_manuals/xilinx13_1/ mb_ref_guide.pdf, 2011. Retrieved on 25-March-2011.

[121] Xilinx. LogiCORE IP block memory generator PG190, v7.3. www.xilinx.com/support/documentation/ip_documentation/blk_mem_ge, 2012. n/v7_3/pg058-blk-mem-gen.pdf Retrieved on 10-September-2013.

[122] Xilinx. Virtex-5 FPGA user guide UG190, v5.4. www.xilinx.com/support/documentation/user_guides/ug190.pdf, 2012. Retrieved on 5-August-2013.

[123] J. Zebchuk, M.Qureshi, V.Srinivasan, and A.Moshovos. A tagless coherence directory. In *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO*, 2009.

[124] H. Zhao, A. Shriraman, and S. Dwarkadas. Space: sharing pattern-based directory coherence for multicore scalability. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques , PACT*, 2010.